Contrôleur de stimulation temps réel embarqué en boucle fermée

Ronan Le Guillou

École Nationale d'Ingénieurs de Brest
Spécialité Systèmes Électroniques Embarqués

8 Septembre 2017
Rapport de stage ingénieur

Contrôleur de stimulation temps réel embarqué en boucle fermée

Ronan Le Guillou

Tuteurs Entreprise : M. SIMON Daniel
M. ANDREU David
Tuteur École : M. NASREDDINE Kamal
Chef d'équipe : Mme. AZEVEDO-COSTE Christine

27 Mars 2017 — 1er Septembre 2017

Remerciements

J'adresse en premier lieu mes remerciements à l'INRIA et à tous ceux qui m'ont permis de réaliser ce stage particulièrement enrichissant.

J'ai eu la chance d'être accueilli pendant ces 5 mois dans la très chaleureuse équipe CAMIN et je souhaiterais avant tout les en remercier vivement. J'ai beaucoup apprécié la richesse de nos échanges.

Je tiens à remercier particulièrement mes maîtres de stage Daniel Simon et David Andreu pour leurs conseils éclairés et leur bienveillance, ainsi que le doctorant Benoît Sijobert pour son aide et le temps qu'il m'a consacré au cours du projet.

Je tiens également à remercier Mr Kamal Nasreddine, mon tuteur ENIB qui s'est assuré avec intérêt du bon déroulement de ce stage.

Et pour finir, je voudrais remercier spécialement ma famille pour son soutien et son aide dans la relecture de ce rapport de stage.

Ronan Le Guillou

Sommaire

Introduction

La recherche progresse dans le domaine assez récent et prometteur des neuro-prothèses. Les prémices de cette discipline ont été posées au début du 20e siècle mais il a fallu attendre un regain d'intérêt dans les années 1970 pour qu'elle prenne son essor. Les entreprises et les équipes de recherche travaillant sur les diverses approches de ce champ d'application sont encore assez rares, particulièrement lorsque les solutions étudiées ont recours à l'implantation chirurgicale. En effet, de nombreuses difficultés doivent être surmontées pour réaliser des essais cliniques et améliorer la technique ainsi que les systèmes développés.

Afin de se rapprocher d'un système utilisable dans la vie de tous les jours par les patients et ainsi continuer à progresser vers la restauration du mouvement, il est crucial pour la recherche dans ce domaine de passer à des prototypes embarqués.

L'équipe-projet CAMIN ("Control of Artificial Movement and Intuitive Neuroprosthesis") de l'INRIA ("Institut National de Recherche en Informatique et en Automatique"), au sein de laquelle ce stage est effectué, focalise ses recherches sur l'étude du mouvement et le développement de solutions neuro-prothétiques utilisant le principe de Stimulation Électrique Fonctionnelle.

La Stimulation Électrique Fonctionnelle, aussi appelée SEF, consiste en la génération de contractions musculaires, sans que celles-ci ne nécessitent un lien direct avec le système nerveux central du sujet. Cette opération peut être réalisée grâce la stimulation électrique sélective du nerf associé au muscle visé ou bien par le recrutement électrique des fibres musculaires situées à la base de celui-ci. Afin de contrôler les contractions musculaires et ainsi pouvoir assister ou restaurer un mouvement, il faut donc disposer d'un système capable de déclencher des stimulations électriques, mais surtout, sachant le faire au moment opportun. Pour cela, l'idéal est de coupler l'utilisation d'une boucle fermée permettant d'avoir des retours physiologiques sur la situation avec des modèles du mouvement afin d'obtenir les meilleures performances possibles.

C'est dans ce cadre et suite au besoin de passer à un système embarqué sur le patient que le projet de ce stage intervient. Il comportera plusieurs étapes nécessaires afin de réaliser un prototype utilisable par l'équipe dans ses divers projets. En effet, pour un système alimentant la stimulation musculaire d'un patient, la stabilité est primordiale. Si la stimulation venait à s'arrêter de façon imprévue lors d'un mouvement, même dans un environnement contrôlé, cela pourrait avoir de graves conséquences comme la chute du patient.

Le but principal de ce projet est de mettre en place un contrôleur embarqué, disposant d'une architecture sans-fil de Stimulation Électrique Fonctionnelle distribuée. Cette architecture sera composée de réseaux de capteurs et de stimulateurs. Le contrôleur devra être capable d'assurer

la gestion de ces réseaux et de les utiliser afin de réaliser une boucle fermée de Stimulation Électrique Fonctionnelle. Cela implique de traiter les données de mesures, fournies par les centrales inertielles HiKoB sur le patient et appliquer la stimulation en conséquence à l'aide des stimulateurs Vivaltis.

Les centrales inertielles, aussi appelées "Unités de Mesures Réparties" (UMR), ainsi que les stimulateurs, appelés "Unités de Stimulation Réparties" (USR) fonctionnent en réseaux. Les passerelles qui permettent de communiquer avec ces capteurs et ces stimulateurs, afin de les contrôler ou d'en récupérer les données, sont en général appelés "Têtes de Réseaux" (TdR).

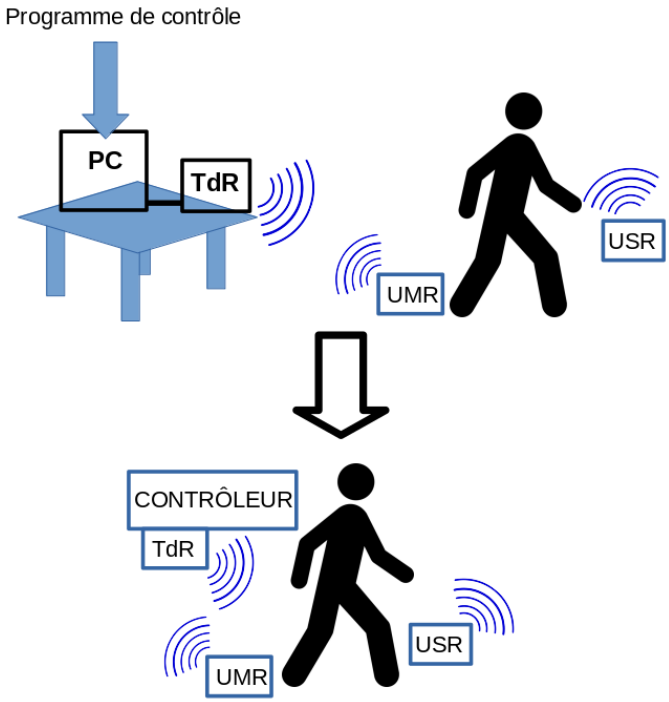


FIGURE 1 – Évolution globale du système de SEF visée par le projet.

Pour atteindre cet objectif il faudra d'abord réaliser le cahier des charges du système puis sélectionner et commander les composants nécessaires (Carte micro-ordinateur, Batterie, etc...). Dans le même temps, un étudiant en Conception Assistée par Ordinateur va créer les boîtiers permettant d'incorporer le système embarqué et de le placer sur le patient. Afin de permettre la réalisation de simulations et d'expérimentations intégrant la carte choisie (Hardware In Loop), un simulateur devra être mis en place sur le système d'exploitation qui aura été choisi. Ce système d'exploitation devra également disposer d'une réactivité suffisante pour héberger un contrôleur de Stimulation Électrique Fonctionnelle. Un noyau Linux modifié sur lequel sera appliqué un patch temps réel devra donc d'abord être mis en place.

Pour finir, le contrôleur de stimulation multi-tâches temps réel en boucle fermée pourra être développé en recueillant les besoins et usages des membres de l'équipe qui seront potentiellement amenés à utiliser ce prototype.

Chapitre 1

Présentation de l'INRIA

L'Institut National de Recherche en Informatique et en Automatique, l'INRIA, vient de fêter son 50ième anniversaire et emploie maintenant plus de 2600 collaborateurs issus d'universités du monde entier. L'institut est organisé en « équipes-projets » qui rassemblent des chercheurs aux compétences complémentaires autour d'un projet scientifique focalisé. Ce modèle ouvert et agile lui permet d'explorer des voies originales avec ses partenaires industriels et académiques. L'INRIA répond ainsi aux enjeux pluridisciplinaires et applicatifs de la transition numérique. A l'origine de nombreuses innovations créatrices de valeur et d'emploi, l'INRIA transfère vers les entreprises (start-up, PME et grands groupes) ses résultats et ses compétences, dans des domaines tels que la santé, les transports, l'énergie, la communication, la sécurité et la protection de la vie privée, la ville intelligente, l'usine du futur, etc... L'INRIA est à l'origine de plusieurs langages de programmation et de divers projets, comme le logiciel libre Scilab.

1.1 Historique

L'histoire de l'Institut National de Recherche en Informatique et en Automatique, l'INRIA, commence le 3 janvier 1967 lors de la création de l'IRIA qui intervient dans le cadre du Plan Calcul. Le Plan Calcul est un plan gouvernemental français qui avait été lancé en fin 1966, avec l'objectif de doter la France des capacités industrielles lui permettant d'être autonome dans le domaine des ordinateurs moyens. L'IRIA a été créé dans le but de développer les capacités de la France dans les domaines des sciences de l'informatique et de l'automatique. Conçu pour unifier et développer la recherche au plus proche des industries et du secteur privé, la formation est l'une de ses priorités. Dès ses débuts, l'institut va organiser des conférences internationales afin d'obtenir une renommée en dehors du territoire français et préparer le terrain pour les coopérations européennes. Celles-ci sont vitales pour faire progresser la recherche sur le plan mondial et pour pérenniser la présence de l'institut dans le paysage de la recherche internationale.

En 1973, l'IRIA s'organise autour du SESORI (Service de synthèse et d'orientation de la recherche en informatique, chargé d'assurer la liaison avec le Plan calcul) et du laboratoire de recherche en informatique et en automatique, le Laboria, mené par Jacques-Louis Lions. Dès 1975, un projet visant à redonner une certaine autonomie aux personnes tétraplégique fut lancé, en collaboration avec l'INSERM, le CNRS et le CEA, le projet SPARTACUS.

En 1979, la décentralisation menace l'existence de l'IRIA : il est question de le délocaliser à Sophia Antipolis ou de le fusionner avec l'IRISA de Rennes (créé avec participation de l'IRIA

en 1975). Finalement, Jacques-Louis Lions obtient le maintien de l'institut à Rocquencourt et celui-ci gagne son « N ». Il sera désormais l'INRIA (décret du 27 décembre 1979). En 1980, la nomination de Jacques-Louis Lions à sa présidence va marquer un tournant pour l'Institut. Avec des moyens limités, il développe un modèle basé sur l'excellence de ses recherches avec un souci constant de transfert vers l'industrie (créations d'entreprises innovantes dans des secteurs stratégiques). L'INRIA est à l'origine d'un réseau de chercheurs européens (ERCIM, European Research Consortium for Informatics and Mathematics) et joue un rôle majeur dans le développement d'Internet en Europe. Après bien des tâtonnements, l'INRIA est désormais doté de missions claires et bénéficie d'une forte réputation internationale.

Depuis 1983, six unités de recherche ont été créées et implantées sur le territoire national. Après avoir fêté ses 40 ans en 2007, l'INRIA reste un institut en pleine expansion. Entre 1999 et 2009, il a doublé ses effectifs. Solidement ancré au sein des écosystèmes industriels et académiques locaux, l'INRIA s'implique toujours plus dans l'espace européen de la recherche. Ouvert sur l'international, il participe au rayonnement des sciences numériques dans le monde. Convaincu que le futur de nos sociétés est numérique, l'INRIA inscrit ses recherches au cœur des grands questionnements sociétaux actuels.

Le transfert pour l'innovation à l'institut prend la forme de dépôts de brevets, de contrats passés avec des industriels, d'animation de Consortia et de soutien aux entreprises innovantes. De 1984 à 2004, 80 entreprises sont créées dont 45 existent toujours. Les collaborations internationales se multiplient.

Des plans stratégiques déclinés en contrats quadriennaux avec l'État, engagent l'INRIA sur des priorités et des performances scientifiques, d'innovation et de transfert, au niveau mondial. Le premier de ces plans stratégiques fut mis en place en 1994.

Le 3 Avril 2017, l'INRIA et l'UNESCO ont signé un accord célébrant la création d'une archive mondiale des logiciels. L'initiative Software Heritage, impulsée par l'INRIA, vise à collecter, organiser, préserver et rendre accessible le code source de tous les logiciels disponibles publiquement.

FIGURE 1.1 – Signature de l'accord entre Antoine Petit, Président-Directeur général de l'INRIA (à gauche) et Irina Bokova Directrice générale de l'UNESCO (à droite) en présence du Président de la République française François Hollande (au centre)

1.2 Structure

1.2.1 Présence sur le territoire national

L'INRIA est présent sur 9 sites en France. Son siège est situé à Rocquencourt, à proximité de Versailles. Les centres historiques sont :

— l'unité de recherche de Rennes Bretagne Atlantique dès 1975. (anciennement Irisa)
— l'unité de recherche Sophia Antipolis Méditerranée (1983).
— l'unité de recherche de Nancy - Grand Est (Lorraine/Loria) (1986).
— l'unité de recherche de Grenoble Rhône-Alpes (1992).

En 2002 fut lancé le programme Inria Futurs, afin d'incuber des centres de recherches et de démarrer l'implantation d'unités de recherches supplémentaires dans l'hypothèse de la poursuite de la croissance de l'INRIA dans la période 2004-2008. Il en résultera l'ouverture des trois sites suivant :

— l'unité de recherche Saclay - Île-de-France (2008).
— l'unité de recherche Bordeaux - Sud-Ouest (2008).
— l'unité de recherche Lille - Nord Europe (2008).

Et pour finir, ouverture d'une nouvelle antenne Parisienne qui deviendra l'unité de recherche Paris en 2009.

FIGURE 1.2 – Présence et implantations de l'INRIA en France

L'INRIA est également présent dans le monde entier, sous forme de partenariat avec des universités, des entreprises et des équipes de recherche dédiées. Cela lui assure de garder un contact privilégié avec un grand nombre de partenaires de recherche à l'international. Ce qui favorise les échanges et permet d'augmenter l'apport de connaissances.

FIGURE 1.3 – Présence et partenariats de l'INRIA dans le monde

1.2.2 Les unités de recherches de l'institut

L'INRIA s'appuie sur un modèle de recherche original qui repose sur une entité de base : l'équipe-projet. Ce modèle est l'élément structurant des activités de recherche de l'institut. Composée d'une vingtaine de personnes, l'équipe-projet(EP) se rassemble autour d'un « leader scientifique », qui établit des objectifs scientifiques sur une thématique approuvée par l'institut. Le chef d'équipe, titulaire d'une habilitation à diriger des recherches, est en charge de la direction et la coordination du travail de l'EP vers ses objectifs. Celle-ci bénéficie d'une large autonomie financière et scientifique, avec un budget constitué de ressources distribuées par le centre et de « ressources propres » générées par les contrats de recherche.

Une EP peut être composée uniquement de chercheurs Inria : c'est l'équipe-projet Inria (EPI). Plus fréquemment, les EP sont associées à des établissements partenaires (universités, écoles, centres de recherche). Ce sont les équipes-projet communes (EPC). Les équipes-projets qui travaillent en partenariat se regroupent pour constituer des Inria Labs.

Une équipe à taille humaine

Les équipes-projet réunissent, autour d'une personnalité scientifique, un groupe de chercheurs, d'enseignants chercheurs, de doctorants et d'ingénieurs. Elles ont toutes un objectif commun : relever un défi scientifique et technologique dans l'un des domaines de recherche prioritaires de l'institut définis dans son plan stratégique.

Un cycle de vie bien défini

Pour obtenir le label « EP », le projet de l'équipe de recherche doit être approuvé par une commission d'évaluation compétente dans son domaine scientifique. Une fois labellisée, l'EP dispose de quatre ans pour mener à bien son programme de recherche et atteindre ses objectifs. Au terme de ces quatre années, l'EP fait à nouveau l'objet d'une évaluation scientifique. Elle peut ainsi être prolongée ou bien arrêtée. La reconduction n'est possible que deux fois, imposant aux Equipe-Projet une durée de vie maximale de douze ans. La durée de vie moyenne est de huit ans.

Une autonomie de gestion

Chaque équipe-projet d'Inria est autonome dans son organisation et sa gestion. L'institut favorise les échanges et les actions collaboratives entre les EP de ses huit centres à travers les Inria Labs. Il facilite aussi les relations des EP avec leurs nombreux pairs internationaux.

Une double mission : sciences & transfert

Au sein d'une équipe-projet, deux objectifs sont poursuivis. La première mission de l'EP est de communiquer le plus largement possible ses résultats scientifiques par le biais de ses publications et en participant à des colloques. La seconde mission de l'EP est de participer activement au transfert des connaissances et des technologies acquises vers l'industrie ou une large communauté d'utilisateurs. Le transfert peut prendre différentes formes : formations, brevets, licences, partenariats stratégiques avec de grands groupes, mise en œuvre de plate-formes technologiques en direction des PME, création d'entreprises...

Un partenaire par essence

La science informatique est par essence « partenariale ». Ainsi, trois équipes-projets sur quatre ont des projets communs avec les divers partenaires de l'institut. Souvent composites, les EP sont parfois hébergées dans un Inria Joint Lab ou encore chez un partenaire : grande école, université, autre organisme de recherche, centre hospitalier universitaire... Les collaborations se font sur la base d'une copropriété des résultats au prorata des moyens affectés, et d'une répartition des efforts, ce qui permet d'optimiser l'impact des recherches.

1.3 Activités

Les axes et thèmes de recherche qui ont été définis par le plan stratégique de l'Inria sont actuellement :

— Mathématiques appliquées, calcul et simulation.

— Algorithmique, programmation, logiciels et architectures.

— Réseaux, systèmes et services, calcul distribué.

— Perception, Cognition, Interaction.

— Santé, biologie et planète numériques.

Ces thèmes sont ensuite sub-divisés en divers domaines de recherche. L'équipe dans laquelle j'ai effectué mon stage fait partie du dernier axe de recherche. Celui-ci se décompose en quatre domaines :

— Sciences de la planète, de l'environnement et de l'énergie.

— Modélisation et commande pour le vivant.

— Biologie numérique.

— Neurosciences et médecine numériques.

C'est dans cette dernière catégorie que se trouve l'équipe CAMIN, ainsi que douze autres équipes-projets de l'institut.

Chapitre 2

Présentation du stage

Ce stage a pour but de développer une solution technologique permettant de rendre entièrement portable par le patient, un système de Stimulation Électrique Fonctionnelle (SEF). Le système utilisé actuellement par l'équipe CAMIN lors de leurs expérimentations sur patients est réparti, grâce à un fonctionnement sans-fil, entre le patient et un ordinateur de contrôle. Mais ce dernier empêche le patient de déambuler sur plus de quelques mètres, ce qui complique les expérimentations et ne permet pas au patient de l'utiliser ailleurs que lors de ces séances. Afin de permettre au patient une plus grande mobilité et de progresser vers un système qu'il pourra emporter chez lui, ce stage vise à déporter le contrôleur de stimulation sur un système portable et autonome.

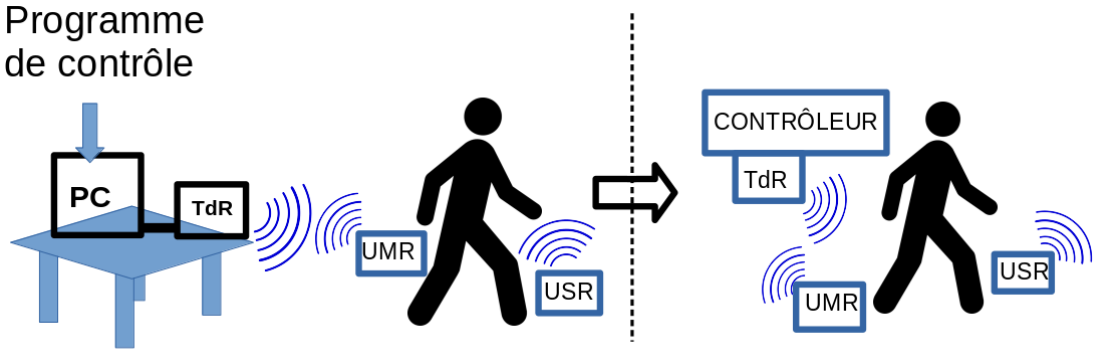


FIGURE 2.1 – Évolution globale du système de SEF visée par le projet.

2.1 Objectifs du stage

2.1.1 Objectifs du projet

L'objectif de ce stage est de réaliser le contrôleur temps réel d'un système de Stimulation Électrique Fonctionnelle (SEF) en boucle fermée. Celui-ci devant être développé avec les contraintes d'un système embarqué temps réel porté par le patient. Il contrôlera, ordonnancera et supervisera la stimulation musculaire du patient en communiquant avec les réseaux de capteurs (Unités de Mesures Réparties) et d'électrodes (Unités de Stimulation Réparties), possédant chacun leurs protocoles de communication propriétaires. Le contrôleur rapatriera en direct les données sur une plateforme séparée (ex : tablette via wifi) qui sera à disposition du praticien pendant les séances avec le patient. Plateforme depuis laquelle la stimulation pourra être configurée ou arrêtée à tous moments.

La réalisation de cet objectif global implique :

— la lecture de documentations diverses, nécessaire pour appréhender le projet dans sa globalité,
— la rédaction du cahier des charges du système,
— la conception du contrôleur et de ses parties matérielles et logicielles,
— la sélection et l'achat du matériel (Ordinateur à carte unique, Batteries, etc...),
— la mise en place sur la carte choisie d'un système d'exploitation (OS) temps réel,
— la mise en place sur la carte choisie du simulateur complet du système de Stimulation Électrique Fonctionnelle afin de permettre des essais avec la partie matérielle intégrée à la boucle de commande (Hardware in Loop),
— le développement en langage C/C++ du contrôleur, communiquant avec les Unités de Mesures Réparties et les Unités de Stimulation Réparties, en collaboration avec l'équipe travaillant sur ce sujet,
— la rédaction de la documentation nécessaire à la récupération et à la pérennité du projet,
— la validation du système entier par simulation puis si possible par expérimentation.

En parallèle, je participerai à la conception du boîtier destiné à intégrer le contrôleur et qui permettra ainsi au patient de porter le système complet.

2.1.2 Évolution des objectifs

Les objectifs présentés dans la proposition du sujet de stage mettaient particulièrement en avant la régulation de la qualité de service du système. La qualité de service englobe plusieurs caractéristiques du système comme les retards et les pertes de trames de communications ainsi que la stabilité et la sécurité des commandes. L'objectif étant d'ajuster en temps réel certaines caractéristiques du réseau de communication, comme la puissance des émissions ou le temps maximum que l'on accepte de passer à attendre les acquittements du Stimulateur Vivaltis par exemple, cela afin de valider l'application d'un nombre de commandes considéré raisonnable et de s'assurer que le système suit les demandes.

Voici les objectifs tels qu'ils étaient présentés sur la proposition de stage :

— Étude de l'architecture existante et de la littérature associée ;

— Conception d'une maquette de contrôleur de SEF portable à partir d'une carte de calcul embarquée existante. Cette carte devra servir de passerelle entre réseaux de stimulateurs et de capteurs, par liaison sans-fil avec les Pods utilisés dans l'équipe ;

— Programmation du contrôleur en C/C++, exécution sur un système Linux embarqué ;

— Proposition de critères de qualité de service pour l'application de commande par SEF tenant compte de contraintes potentiellement conflictuelles entre commandes et utilisation des ressources et intégration de contraintes liées à la sureté de fonctionnement, caractérisation de la qualité de service apporté par des formats de trames modifiées pour implémenter les services nouveaux ;

— Conception et implémentation du contrôleur d'ordonnancement et des contrôleurs de SEF mettant en œuvre la Qualité de Service. Les fonctions de commande pourront être en ligne (feedback), pour autant que les ressources de calcul soient suffisantes. Des solutions partiellement hors ligne pourront aussi être envisagées.

Cependant, la qualité de service s'est avérée être difficile à réguler puisque les réseaux de capteurs et de stimulateurs étant propriétaires, nous n'avons accès qu'aux fonctionnalités présentées dans leurs protocoles clients. Les possibilités de configurations autorisées par leurs protocoles respectifs sont assez restreintes. Par conséquent les modifications en temps réel des caractéristiques de communication ou du réseau dans son ensemble, afin de suivre les besoins du système, sont très limitées.

Les librairies existantes, permettant d'analyser les données envoyées par la tête de réseau HiKoB au contrôleur, ainsi que les scripts effectuant les calculs sur ces données, sont écrits en langage Python. Quant aux fonctions permettant de communiquer avec la tête de réseau Vivaltis, elles étaient écrites en langage Matlab.

Afin de ne pas avoir à traduire les librairies HiKoB en langage C++ ou de recourir au lancement de scripts python dans un programme écrit en C++, le langage Python s'est imposé pour le développement du contrôleur. De plus, l'écriture de fonctions équivalentes au code Matlab est plus aisée et rapide en langage Python qu'en langage C++.

Toutefois, un interpréteur Python, servant d'intermédiaire entre le code du contrôleur et le système d'exploitation, crée un niveau d'abstraction supplémentaire, traduisant le code Python en code exécutable. Ce choix nous prive d'un contrôle précis et bas niveau des tâches qui sont réellement effectuées sur le système. Cela entraine inévitablement une perte d'optimisation au niveau des temps de calculs et d'exécution, points cruciaux pour un système temps réel, qui plus est dans le domaine médical. Le point réellement bloquant au niveau temporel est cependant la commande du réseau de stimulateurs et l'attente des acquittements de trames. Le contrôleur pourra toujours être traduit en C++ ultérieurement si le gain d'optimisation et de contrôle sont nécessaires.

De fait, les objectifs ont été légèrement modifiés pour se concentrer sur le contrôleur lui-même, à la fois sur les aspects matériels et logiciels ainsi que son intégration dans un boîtier pour le rendre portable par un patient.

2.2 Contexte du stage

2.2.1 L'équipe CAMIN

L'équipe "Control of Artificial Movement and Intuitive Neuroprosthesis" (CAMIN), dirigée par Christine Azevedo-Coste est dédiée à la conception et au développement de solutions neuro-prothétiques réalistes dans le contexte de déficiences sensori-motrices en interaction avec des partenaires cliniques. Leurs efforts sont concentrés sur l'objectif d'avoir un impact clinique : l'amélioration de l'évaluation fonctionnelle du patient et/ou de sa qualité de vie. Le mouvement est au centre de leurs travaux de recherche. L'exploration et la compréhension de son origine, ainsi que son contrôle, constituent l'un de leurs deux principaux axes de recherche. En effet, afin de faire avancer les solutions neuroprothétiques, il est nécessaire d'améliorer la connaissance sur les rôles des systèmes nerveux, périphérique et central, dans le contrôle du mouvement. En se basant sur les résultats du premier axe, des neuro-prothèses sont déployées, le second axe étant l'assistance et/ou la restauration du mouvement.

Voici l'adresse du site internet de l'équipe CAMIN :

http ://www.lirmm.fr/camin/

Vous trouverez ci-dessous le schéma synthétique représentant leur approche sur le sujet de la mise en place de solutions neuro-prothétiques :

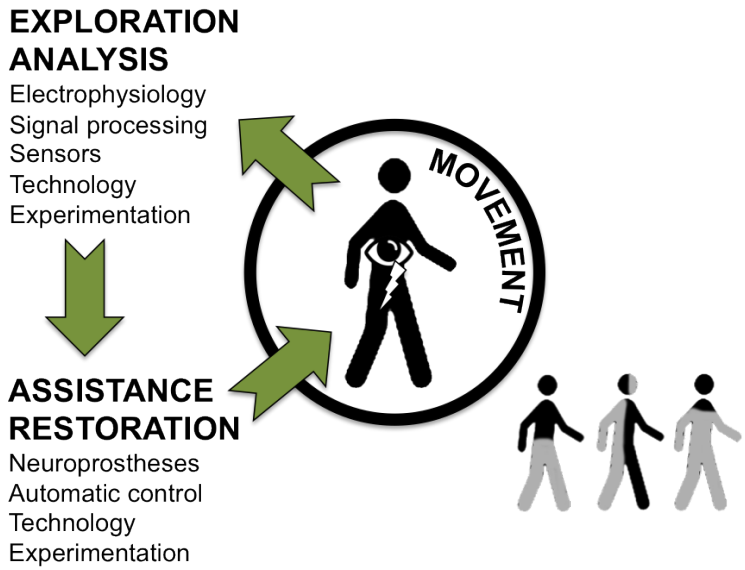


FIGURE 2.2 – CAMIN approach to Functionnal Electrical Stimulation

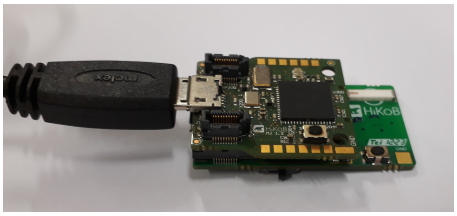
Cette équipe et quelques autres de l'INRIA basées sur Montpellier, sont hébergées par le Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM).

2.2.2 Matériel utilisé

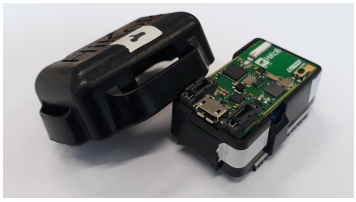
2.2.2.1 Matériel médical

Plusieurs projets de l'équipe CAMIN utilisent du matériel médical propriétaire. Certains équipements ont d'ailleurs été développés au sein de l'équipe puis récupérés par des entreprises comme Vivaltis. Ce matériel de pointe, sans concurrent direct en technologie sans fils dans son domaine d'application, est un invariant du projet et doit être intégré avec soin au contrôleur. Seront utilisés dans ce projet deux principaux systèmes, des centrales inertielles, ou Unités de Mesures Réparties (UMR), afin d'avoir un retour physique (feedback) du patient pour la boucle de contrôle et des pods de stimulation, ou Unités de Stimulation Réparties (USR), permettant de délivrer la stimulation électrique au patient.

Les centrales inertielles sont développées par la société HiKoB. Celles-ci sont appelées nœuds HIKOB Fox, elles prennent en compte 9 degrés de libertés grâce à un Accéléromètre 3D, un Gyroscope 3D et un Magnétomètre 3D. Elles possèdent également une sonde de pression, de température et une petite batterie. La tête de réseau elle, ne sera pas sur batterie mais alimenté par sa connexion USB avec le contrôleur.



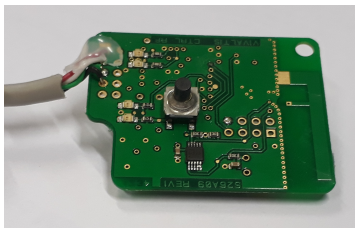
(a) Tête de réseau HiKoB



(b) Centrale inertielle HiKoB Fox (4cm x 2cm x 2cm)

FIGURE 2.3 – Élements du réseau HiKoB.

Les stimulateurs sont développés par la société Montpellieraine Vivaltis. Ceux qui sont utilisés pour ce projet sont les PODs Universels Stim/Bio PHENIX USB Néo. Ils communiquent par liaison sans fil avec une unité centrale qui sera elle aussi connectée au contrôleur via USB. Chaque POD Stim/Bio comporte 2 voies de biofeedback(ie : mesure de l'activité Electro-Myo-Graphique "EMG" d'un muscle) et 2 voies de stimulation qui permettront de stimuler indépendamment deux muscles distincts.



(a) Tête de réseau Vivaltis

(b) Stimulateur Vivaltis (8cm x 5cm x 4cm)

FIGURE 2.4 – Élements du réseau Vivaltis.

2.2.2.2 Matériel de développement

Lors de ce stage j'ai principalement utilisé des systèmes d'exploitation basés sur des distributions Linux, cependant afin de s'assurer du bon fonctionnement multi-plateformes du contrôleur et de sa facilité d'utilisation, j'ai également utilisé des ordinateurs sous Windows. A mon arrivée dans l'équipe nous avons installé Fedora 24 sur un ordinateur libre que j'ai utilisé par la suite comme poste de travail. Afin de vérifier le fonctionnement du contrôleur avec une communication sans fil, j'ai également installé Fedora 24 sur un ordinateur personnel sur lequel j'ai par la suite rédigé mon rapport de stage synchronisé grâce au logiciel de gestion de versions décentralisé Subversion.

La plateforme utilisée pour le contrôleur est la carte Raspberry Pi 3 modèle B. Je reviendrai sur ce choix et les critères de sélection ayant menés à cette décision dans la sous-partie 3.1.1. J'ai ensuite naturellement choisi d'utiliser le système d'exploitation Raspbian, développé spécifiquement pour les cartes Raspberry Pi et permettant un support aussi stable que possible des fonctionnalités et des capacités qu'elles offrent. Raspbian est une distribution basée sur Debian, maintenue par la fondation Raspberry Pi.

La carte Raspberry Pi 3, sortie en Février 2016 remplace la Raspberry Pi 2 Modèle B. Elle a un processeur Broadcom BCM2837 64 bits 1,2GHz de quatre cœurs. Cependant, la version du système d'exploitation utilisée lors de ce stage ne prend pas encore en charge les fonctionnalités 64bits et fonctionne donc dans un mode de compatibilité ARMv7 32bits. Cette carte dispose entre autres d'une mémoire RAM de 1 Go, d'une puce intégrée de communications sans fil, de 4 ports USB et d'une barrette de 40 connecteurs GPIOs. Ces entrées et sorties analogiques à usage général permettent d'utiliser divers accessoires (boutons, buzzers, leds ...). Elles permettent aussi d'alimenter directement la carte en tension 5V en courant direct, par exemple via une batterie. Dans ce type d'utilisation, des précautions sont à observer, comme l'ajout d'un fusible ou d'un régulateur de tensions pour éviter des problèmes d'alimentation.

Ne disposant pas de mémoire Flash, le système d'exploitation et les données sont stockés sur une carte microSD. Elle est de la taille d'une carte de crédit (85,60 mm x 53,98 mm x 17 mm) et a une consommation relativement basse. L'horloge principale de son processeur fonctionne par défaut sur une bande de fréquence allant de 0,6 GHz à 1,2 GHz

2.3 Gestion du projet

2.3.1 Méthode de travail

Afin d'avoir une base suffisante de connaissances dans le domaine d'activité de l'équipe et sur les travaux réalisés en amont sur le simulateur du système de Stimulation Électrique Fonctionnelle, la première étape du stage, nécessaire pour interagir avec les collègues dans un contexte pluri-disciplinaire, fut de lire de la documentation, des articles scientifiques sur les travaux réalisés et des rapports rédigés par des stagiaires précédents.

Ce projet, réalisé dans un contexte de collaboration multidisciplinaire, est destiné à servir de prototype dans une équipe de recherche en contact avec des patients, des médecins et des entreprises. Le développement de ce prototype visant à embarquer un système de stimulation, m'a poussé à recueillir les besoins et usages des personnes utilisant le modèle actuel. J'ai

donc échangé régulièrement avec l'équipe CAMIN et plus particulièrement le doctorant Benoît Sijobert, travaillant sur le protocole médical de stimulation électrique fonctionnelle en cours d'essais cliniques.

Une méthode agile s'est imposée naturellement afin d'intégrer réellement les propositions et les besoins de l'équipe au processus de développement, à fortiori dans ce contexte de recherche.

Le contrôleur développé étant destiné à être une base de prototypage, il est également essentiel de garder une vue globale du projet pour permettre autant que possible l'ouverture et l'accessibilité du système. Des réunions régulières ont permis de mettre en évidence des fonctionnalités manquantes ou des priorités à réajuster.

J'ai participé à l'encadrement de l'étudiant en Conception Assisté par Ordinateur qui est venu, dans le cadre d'un stage de deux mois, pour concevoir les boîtiers de ce projet. J'ai donc échangé quotidiennement avec lui afin qu'il puisse intégrer, avec le plus de précision possible, tous les composants du contrôleur dans les boîtiers.

2.3.2 Planning du projet

Afin de visualiser les différentes temporalités en jeu dans ce projet, le diagramme de Gantt ci-dessous présente les diverses tâches effectuées et les durées sur lesquelles elles ont étés réalisées.

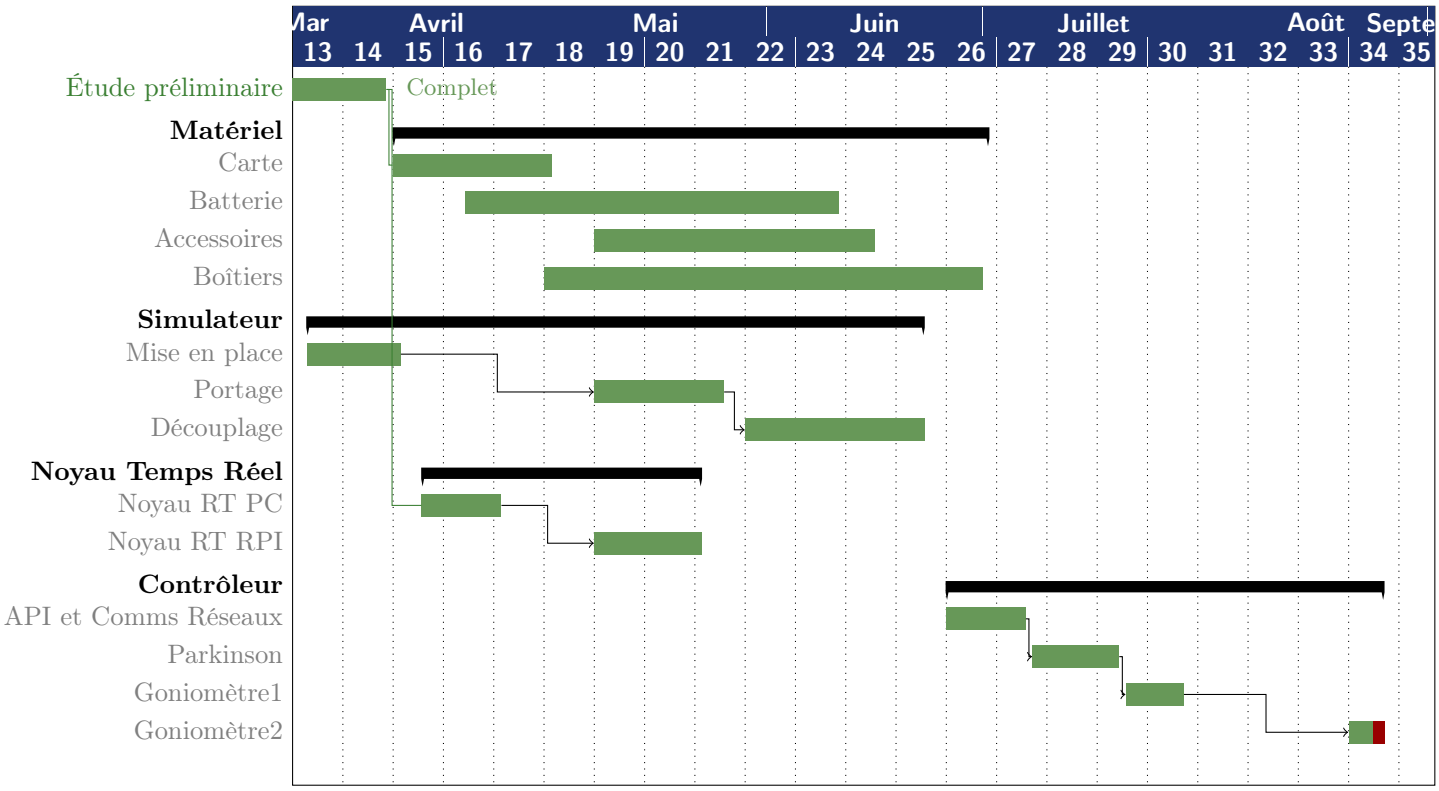


FIGURE 2.5 – Diagramme de Gantt du projet.

Chapitre 3

Réalisation du projet

Le contrôleur qui va être réalisé devra premièrement être interfacé avec l'architecture matérielle de Stimulation Électrique Fonctionnelle, c'est à dire qu'il devra pouvoir communiquer avec les Têtes de réseaux (ou passerelles) HiKoB et Vivaltis et les commander. Ces passerelles utilisant des connectiques USB, devront pouvoir être branchées au contrôleur. Deuxièmement, il devra disposer d'entrées sorties à utilité générale (GPIO) afin de permettre l'utilisation d'outils de mesures ou d'interventions extérieures que l'équipe pourrait utiliser. Troisièmement, le système devra disposer d'une connectivité WiFi pour ne pas être isolé en mode autonome. Quatrièmement, il devra pouvoir être alimenté grâce à une batterie pour être réellement embarquable, avec une autonomie d'au moins 6 heures et de 8 dans l'idéal. Finalement, le contrôleur qui sera développé ayant pour ambition d'être une base de prototypage pour les années à venir il devra donc pouvoir accueillir des algorithmes potentiellement de plus en plus complexes et des réseaux de capteurs et de stimulateurs de plus en plus étendus.

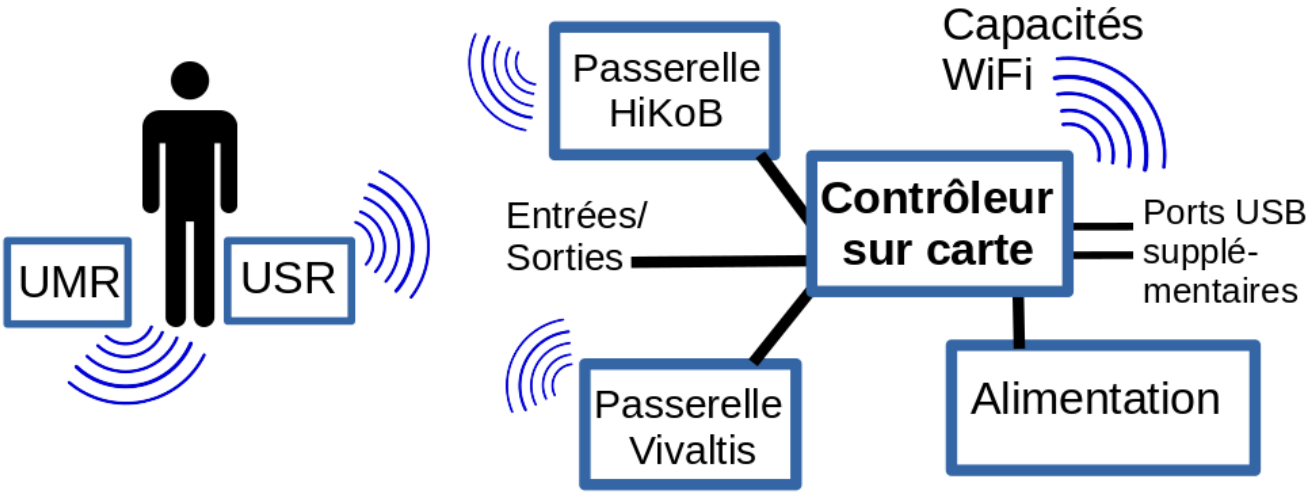


FIGURE 3.1 – Représentation du système matériel cible.

3.1 Partie Matérielle

3.1.1 Sélection de la carte

3.1.1.1 Critères de sélection de la carte

Rapidement après avoir posé les objectifs du projet, nous avons discuté des spécifications nécessaires pour les atteindre. Le prototype doit être réutilisable pendant plusieurs années et il faut pouvoir augmenter ultérieurement la complexité des réseaux gérés par le contrôleur. Pour cela, il est donc nécessaire d'avoir une puissance de calcul suffisante, tout en restant dans une gamme de consommation raisonnable et de disposer de plusieurs ports USB afin d'ajouter des Têtes de Réseaux si besoin. L'ajout de matériel gérés également par USB étant une possibilité, nous avons estimé que 4 ports USB serait l'idéal.

La présence d'entrées sorties à utilité générale (General Purpose Input/Output) est essentielle. Elles permettront d'utiliser différents types de boutons, des accessoires comme un buzzer et d'envisager d'autres interventions extérieures comme des dalles de pression pour détecter les mouvements et variations d'effort au niveau des pieds du patient.

Il était également important d'avoir un contact sans fil avec une autre plateforme de contrôle ou de supervision, comme un ordinateur portable utilisé par le praticien ou le chercheur pendant les sessions. La présence d'un module WiFi intégré dans la carte s'est donc montrée être un critère principal supplémentaire. Car même si un dongle wifi connecté en USB au contrôleur pourrait suffire, il occuperait un port USB.

Le système étant prévu pour être embarqué sur un patient, les critères de poids et de taille de la carte étaient également importants. La plupart des cartes dites "Single Board Computer" disponibles sur le marché sont au format taille "carte de crédit" ou "Credit-Card-Size" et pèsent moins de 50 grammes, ce qui est parfait pour l'application souhaitée.

Il m'est apparu préférable d'orienter les recherches vers des solutions ouvertes ou semi-ouvertes disposant d'un support technique actif et durable, ou du moins d'une communauté de développeurs active et pérenne. En effet il est nécessaire de pouvoir se documenter facilement ou d'obtenir de l'aide sur les spécificités de la mise en place d'applications dans des contextes peu courants. Les patchs temps réel de noyaux Linux par exemple ne sont pas disponibles sur toutes les plateformes. C'est une communauté de développeurs qui les porte sur de nouvelles plateformes et les maintient à jour.

Pour récapituler, les critères de sélection de la carte micro-ordinateur sont :

1. la présence de la communauté/support (Pérennisation),
2. la puissance de calcul (Souplesse de traitement),
3. la consommation d'énergie (Autonomie),
4. le nombre de ports USB (Possibilité de complexification),
5. la présence d'un module Wifi intégré (Simplicité de communication),
6. le volume et le poids (Embarquabilité).
7. la présence d'une barrette GPIO (Multi-fonctionnalité).

3.1.1.2 Choix de la carte

Suite à la mise en place de ces critères et du cahier des charges, je me suis mis à étudier les propositions sur le marché des cartes de développement et des ordinateurs en carte unique grand public. Je suis arrivé au constat que l'on peut distinguer les cartes dites "industrielles" (Phytec et Technologic Systems ex :TS7260) et "les grand public" (Raspberry Pi, Beagleboard et Odroid). Les solutions industrielles sont nécessaires lorsque les contraintes physiques ou environnementales du système sont particulières (pression, température, champs magnétiques, etc...). Les cartes grand public offrent plus de souplesse d'utilisation, sont moins confidentielles et permettent de bénéficier d'une communauté active de développeurs.

Ce choix sensible m'a amené à échanger avec notamment deux professeurs en électronique à l'ENIB, Mr Boucharé et Mr Kerhoas, échanges qui ont tous convergé vers la même conclusion. Sur tous les produits du marché actuel, d'après les critères de ce projet, l'idéal serait Beagleboard ou Raspberry Pi. On m'a mis en garde sur le support incertain de patchs temps réel comme RT PREEMPT. Cependant après quelques recherches j'ai trouvé des articles détaillant la mise en place de tels patchs par des particuliers et des développeurs, attestant de la faisabilité de l'application d'un patch temps réel sur le noyau Linux 4.9.

J'ai alors proposé une réunion afin de discuter de ces différentes options. Pour étayer ma présentation, j'ai synthétisé mes recherches sous forme d'un tableur permettant de comparer les caractéristiques des produits nous intéressant, tels que leurs prix, leurs puissances de calcul, leurs consommations, leurs nombres de ports USB accessibles, le dynamisme de leurs communautés etc... Les images 3.2 et 3.3 ci-dessous montrent la partie de ce tableur comparant les principales candidates et un test de consommations énergétique de certaines de ces cartes..

Après discussion avec mes tuteurs, le choix s'est dirigé vers les cartes BeagleBone Black Wireless, BeagleBone Green Wireless et la Raspberry Pi 3 Modèle B. Toutes trois sont relativement similaires et ont un module wifi intégré. Cependant la BeagleBone Black Wireless ne possède qu'un port USB et la BeagleBone Green Wireless perd ses ports Ethernet et HDMI afin de fournir 4 ports USB. Ces deux choix assez contraignants n'auraient pas permis une utilisation aisée et directe. La carte Raspberry Pi 3 Modèle B, elle, disposait de 4 ports USB, d'un port Ethernet et d'un port HDMI, tout en ayant une communauté un peu plus étendue et active que les cartes BeagleBoard. De plus, plusieurs sources concordantes montrent, comme la figure 3.4, l'amélioration des performances de la nouvelle version du Raspberry Pi modèle B au niveau de sa consommation. C'est donc cette carte (Voir image 3.5) qui nous a paru la meilleure candidate par rapport aux critères du projet.

	Raspberry Pi				Beagle	
	3 B	B+	A+	Zero W	Bone Black W	Bone Green W
CPU	ARM710	ARM11	ARM11	ARM11	TI Sitara AM3358	TI Sitara AM3358
Nb Cores	4xA53	1	1	1	1 x A8	1 x A8
Frequency	1.2GHz	700MHz	700 MHz	1GHz	1GHz	1GHz
RAM	1Go	512Mo	512 Mo	512Mo	512Mo	512Mo
ROM(Flash)	✘	✘	✘	✘	4GB	4GB
Built-in Wifi	✔ +ethernet	✘ +ethernet	✘	✔	✔	✔
USB Ports	4+1	4+1	1	1	1+1	4
Micro USB	✔	✔	✔	✔	✔	✔
Alim DC	5V	5V	5V	5V	5V	5V usb only
Current ~ mA	720	330	230	280	300	300
Power ~	4 W	3 W	1W	1.1W	1.5 W	1.5 W
Size mm	85 x 56 x 17	85 x 56 x 17	65 × 54 × 17	65 x 30 x 5	85 x 53 x 4	86 x 53 x 4
HDMI port	✔	✔	✔	mini	✔	✘
Micro SD	✔	✔	✔	✔	✔	✔
Price ~	42 euros	36 euros	26 euros	10$	64 euros	45 euros
					2 PRU 32bits microC 200 MHz	
	45g	45g	23g	9g	42g	42g

FIGURE 3.2 – Comparaison de plusieurs cartes disponibles.

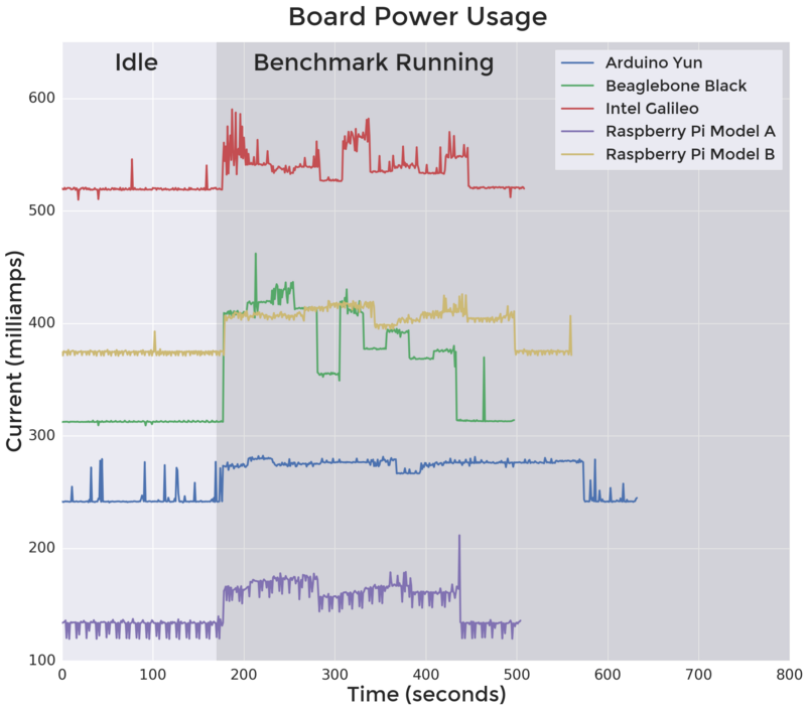


FIGURE 3.3 – Consommation d'énergie de cartes de plusieurs fabricants

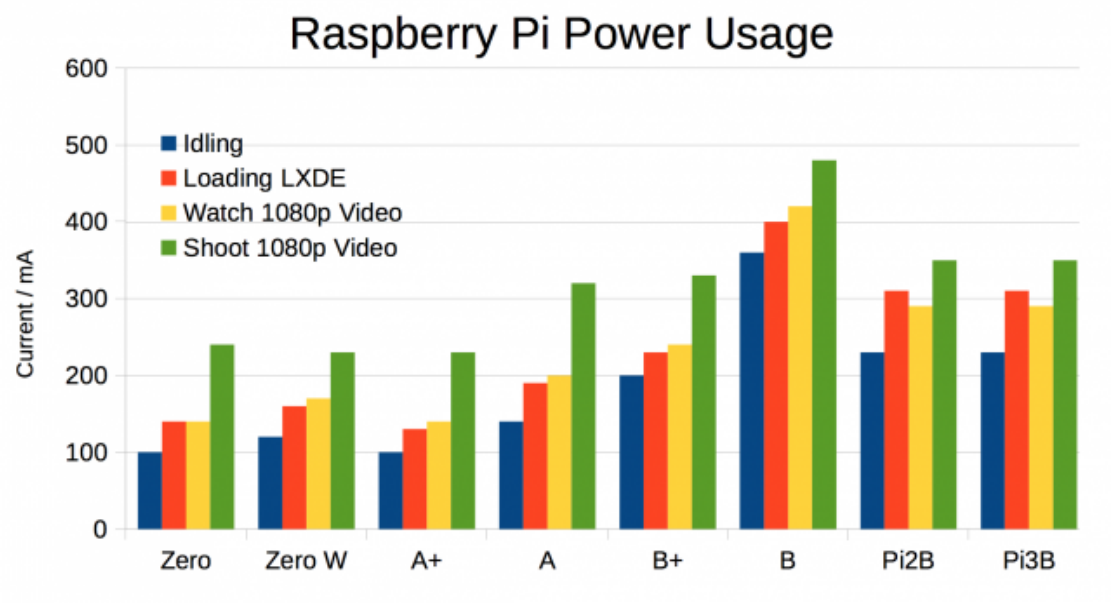


FIGURE 3.4 – Consommation d'énergie des cartes Raspberry Pi

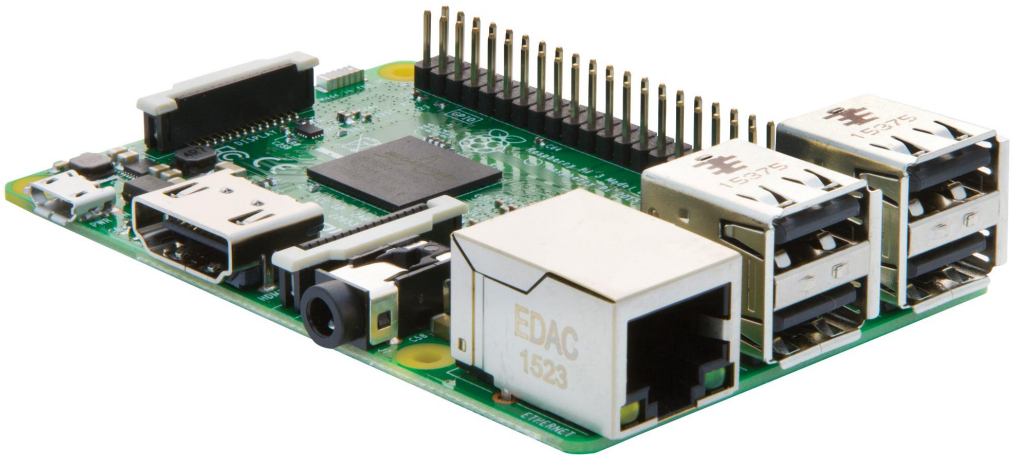


FIGURE 3.5 – Raspberry Pi 3 modèle B, version sortie en Janvier 2016

3.1.2 Sélection de la batterie

N'ayant pas d'instruction particulières concernant le type de batterie à commander, je me suis mis à rechercher des batteries externes ou "Power Banks" disponibles sur internet. Celle-ci présentaient de très bonnes capacités à des prix assez bas. La plupart étaient assez volumineuses mais certaines pouvaient correspondre aux contraintes de dimensions. Ces batteries sont en général normalisées à une tension de sortie de 5.0 V afin de charger les appareils comme les téléphones portables. Cela aurait pu convenir à l'alimentation de la carte Raspberry Pi car elle accepte en entrée, d'après ses caractéristiques affichées, des tensions allant de 4.8 volts à 5.2 volts. On peut trouver assez facilement des batteries de commerce allant jusqu'à 26 800 mAh soit, à 5 V de sortie en moyenne, 134 Wh. Ce qui est considérable, mais rarement approché en réalité.

Nous envisagions à cette période de ne réaliser qu'un seul boîtier assez long, comportant le contrôleur principal, les deux têtes de réseaux HiKoB et Vivaltis, ainsi que la batterie en dessous des composants précédemment cités.

Cependant, ces batteries de commerce n'ont pas un grand nombre de cycles d'utilisation avant que la capacité maximum ne chute. Leur capacité maximum de charge, dès la réception, est d'ailleurs rarement proche du montant présenté. De plus la tension de sortie n'est pas particulièrement stable et peut fluctuer significativement, elles ne sont donc en général pas idéales pour des applications d'alimentation continue de systèmes mais plutôt utilisées pour recharger des appareils. Dans le cadre d'un système alimentant la stimulation musculaire d'un patient, la stabilité du système est primordiale. Si la stimulation s'arrêtait lors d'un mouvement, cela pourrait entrainer la chute du patient et donc avoir de graves conséquences.

Après discussions avec mes tuteurs, la recherche de batteries se dirigea vers les produits de fournisseurs industriels, telles que des batteries aux normes militaires ou médicales.

Les sessions d'expérimentations du protocole clinique durent une journée, soit environ 8 heures durant lesquelles le système est utilisé sur un groupe de patients, les uns après les autres, désigné cliniquement comme une cohorte de patients. La consommation totale du système étant évaluée avec une petite marge à environ 1,0 Ah en ajoutant les consommations des différents modules du système :

1. 700 mA pour la carte Raspberry Pi,
2. 50 mA pour les connecteurs GPIO et l'alimentation des accessoires y étant reliés,
3. ainsi que environ deux fois 80 mA pour les têtes de réseaux HiKoB et Vivaltis.

Afin de permettre au système embarqué de fonctionner durant une session entière de 8 heures, pour une tension d'entrée de 5 V il faudrait une capacité énergétique de 40 Wh.
(Puissance (en Watt-heure) = Voltage (en volts) x Consommation (en Ampères par heure))

Les cellules individuelles étant en général de 3.4 V, les batteries de 5 V sont très rares dans le milieu industriel. Les seules disponibles n'étaient pas adaptées à notre application. Il était donc dès lors inévitable de passer par un régulateur de tension DC-DC type convertisseur Buck (convertisseur "step-down"). Cela engrangeant une perte de puissance de 10% au maximum d'après les convertisseurs et régulateurs disponibles actuellement, la capacité recherchée passe donc à 44 Wh. Avec l'aide d'Olivier Climent, l'ingénieur qualité de l'équipe, nous avons trouvé

une batterie correspondant aux critères recherchés, la batterie "Standard Li-ion Smart Battery Pack RRC2054" produite par RRC Power Solutions.

Cette batterie paraissait une bonne candidate :

— Sa capacité de 48Wh (15Volts et 3.2Ah), légèrement supérieure à celle recherchée.

— Ses dimensions (85.4mm par 77.6mm par 23mm) assez proche de la carte Raspberry Pi.

Mais deux caractéristiques pouvaient s'avérer problématique, sa largeur importante et son poids un peu élevé (240 grammes).

Lors d'une réunion d'avancement nous avons conclu qu'une batterie comme celle-ci serait trop lourde et volumineuse pour que le système soit aisément portable par le patient. Une batterie moins large et plus légère était donc nécessaire. Après de nouvelles recherches, c'est la batterie "Standard Li-ion Smart Battery Pack RRC2040" qui est apparue comme la meilleure solution. Ces réductions en poids et en largeur venant obligatoirement au prix d'une réduction en capacité énergétique : 2.95Ah à 11.25V soit environ 33Wh. Avec une perte de 10% due à la présence d'un convertisseur/régulateur DC-DC, nous pouvons supposer une capacité de 30Wh. Cela correspond à 6Ah, ce qui suppose qu'avec une consommation du système de 1Ah, l'autonomie sera de 6 heures. Ayant fait l'estimation de la consommation en prenant une petite marge, on peut espérer atteindre l'objectif des 8 heures si la consommation de la carte Raspberry Pi n'est pas excessive.

FIGURE 3.6 – Batterie RRC 2040 de 33.2Wh

Cette batterie correspondant mieux aux différentes attentes, j'ai pris contact avec la société Avnet Abacus. J'ai réalisé la commande mi-Juin, mais la batterie n'est finalement arrivée que début Août pour des raisons inexpliquées par l'entreprise. Cette attente a mis en pause la réalisation des boîtiers car il était nécessaire d'avoir les batteries et les barrettes de connecteurs appropriées à ce modèle pour les terminer. Les barrettes de connecteurs, elles, ont été commandées sur un autre site, en raison de la nécessité de les acheter par paquets de 100 chez le fournisseur Avnet Abacus.

3.1.3 Sélection des composants

La sélection des composants et la préparation des commandes s'est faite sur une période assez étendue. La procédure pour passer une commande de matériel n'étant pas anodine, il était préférable de grouper les commandes. Le choix de chaque composant s'est fait en recherchant attentivement les offres sur les sites de fournisseurs de matériel électronique. Pour chaque composant, une attention particulière était portée à la sélection du design le plus petit et pratique possible, afin d'éviter d'avoir à agrandir les boîtiers du système.

3.1.3.1 Choix du câble d'alimentation

Choisir le câble d'alimentation nécessitait de fixer précisément les besoins en alimentation, c'est à dire la puissance que ce câble devra transporter, sachant qu'il sera au contact du patient, allant d'un boîtier à l'autre. Il fallait aussi définir le nombre de fils conducteurs présents dans ce câble, car il détermine la quantité d'informations que nous pouvons récupérer auprès d'une batterie intelligente comme la RRC2040. Il est en effet possible d'obtenir la température de la batterie, son état de charge ainsi que de configurer certains comportements. Ce devait donc être un câble ni trop gros ni trop rigide, mais blindé, afin d'éviter de trop grandes perturbations électro-magnétiques, par rapport à des appareils médicaux pouvant se trouver à proximité lors d'utilisation dans des hôpitaux. Après avoir étudié les calibres disponibles, il fut décidé de prendre un câble multi-conducteur blindé Alpha Wire 22AWG à 3 conducteurs, d'un ampérage nominal maximal d'environ 3 Ampères.

Pour utiliser ce câble nous avons choisi des embases à fixer aux boîtiers. Après avoir découvert les différentes possibilités, nous nous sommes dirigés vers des jeux d'embases mâles et femelles adaptés à la taille du câble choisi, montés sur panneau par serrage d'un écrou et se verrouillant après insertion l'un dans l'autre par un quart de tour. Ce type de connecteurs est appelé "Embase à fermeture à baïonnette".

Afin d'éviter tout contact physique direct avec le connecteur d'alimentation, l'embase qui sera montée sur le boîtier de la batterie sera une embase femelle dont les contacts sont internes. Le câble d'alimentation viendra s'y insérer via une embase mâle à montage sur câble et l'autre coté du câble sera terminé par une embase femelle de nouveau, à montage sur câble, afin d'éviter tout contact. Cette embase femelle viendra finalement se loger sur une embase mâle à montage sur panneau, situé à l'arrière du boîtier contrôleur du système. Ces connecteurs sont assez courts et ont un diamètre maximum de 12 millimètres. Ce qui nous permet de les placer sur le boitier en encombrant le moins possible les façades ainsi que l'intérieur des boîtiers. Beaucoup de produits similaires étaient particulièrement volumineux en diamètre du trou en façade ou en longueur. Nous prendrons également un peu de câble thermo-rétractable doté d'un bon coefficient d'élasticité afin de pouvoir bien isoler les connecteurs si nécessaire.

3.1.3.2 Composants du boîtier contrôleur

Sélection de la LED

Au fur et à mesure des discussions avec Benoît Sijobert et mes tuteurs, nous décidions d'ajouter des composants, comme une led permettant de fournir un signal visuel, pouvant

être configuré pour différents contextes. Les modes de fonctionnement imaginés pour cette led sont un clignotement rapide lorsque la stimulation électrique est en cours, un clignotement lent lorsque le programme est lancé mais que la stimulation n'est pas encore délivrée et finalement, automatiquement à l'allumage du système d'exploitation de la carte, pleinement allumée en continu pour signifier que le Raspberry Pi a bien démarré. La led choisie est faite pour être montée en façade, grâce à un système d'écrou et de corps fileté. Elle est également d'un design évitant toute possibilité d'accrochage éventuel avec les vêtements du patient. Elle doit être visible non seulement de face mais aussi de coté.

Sélection du Buzzer

Pour renforcer le signal visuel par un signal sonore ou pour apporter d'autres informations, l'ajout d'un buzzer fut également mis en avant. J'ai donc testé le fonctionnement de différents buzzers et composants piezo-électriques. Après avoir regardé un grand nombre de produits de cette gamme, nous avons opté pour un buzzer particulièrement petit (14 mm de diamètre et 8 mm de hauteur), acceptant une grande marge de tensions en entrée (1,5 VDC à 16 VDC) et disposant d'un bon volume sonore par rapport à sa consommation maximale de 8 mA. En effet il est caractérisé par une pression acoustique en décibels de 85dB minimum lorsque alimenté en 12 VDC. Les connecteurs GPIO de la carte sont limités en courant maximum de sortie. Chaque connecteur est individuellement restreint à 15 mA maximum et seuls 50mA peuvent être délivrés en même temps sur les sorties de la barrette de connecteurs GPIO.

Sélection des embases Jack 3,5mm

Il est apparu judicieux aux chercheurs de l'équipe d'ajouter des boutons pour permettre aux patients de transmettre des informations au système, ou de les laisser eux même déclencher la stimulation lorsqu'ils souhaitent entamer un mouvement. Certains boutons utilisés par l'équipe, ont une connectique Jack 3,5 mm. Il nous est donc paru nécessaire de placer 3 entrées Jack 3,5mm sur différentes façades du boîtier contrôleur. Après quelques recherches et discussions, j'ai trouvé un connecteur Jack femelle Lumberg KLB 4, 3 pôles, pour montage sur panneau via serrage par écrou, rond, d'un diamètre maximum assez petit (8mm) et peu profond (2cm).

Sélection d'un connecteur Micro-USB mâle

La tête de réseau HiKoB est munie d'une connectique micro-USB type B. Afin de l'installer dans le boîtier, j'ai cherché des fiches mâles, les plus minimalistes possible. J'ai trouvé un connecteur dénudé de 14 millimètres de long, à souder directement sur câbles.

(a) Led (b) Buzzer (c) Embase Jack 3,5mm (d) Fiche Micro-USB

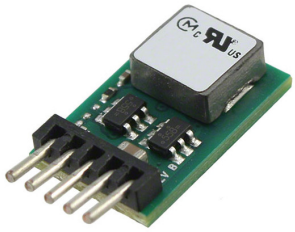
FIGURE 3.7 – Composants du boîtier contrôleur.

3.1.3.3 Composants du boîtier d'alimentation

Afin de passer des 11.25 Volts de la batterie sélectionnée aux 5 Volts stables nécessaires à l'alimentation continue de la carte, il me fallait trouver un régulateur convertisseur de tensions de courant direct à courant direct. Après des recherches avec Olivier Climent, l'ingénieur qualité de l'équipe, nous avons choisi le Convertisseur DC/DC OKR-T/3-W12-C de Murata Power Solutions. Il prend en entrée une gamme de tensions allant de 4,5 V à 14 V, a une efficacité de 93% et peut délivrer jusqu'à 3 Ampères. Ce convertisseur est réglable par la mise en place d'une résistance entre ses bornes Trim et Ground. Il est conseillé d'après la datasheet du composant d'utiliser une résistance de 268 Ohms environ pour une tension de sortie de 5 V. Nous avons donc également commandé quelques résistances à puces CMS de cette valeur.

Ce régulateur intègre d'après sa datasheet un système de lissage de la sortie et l'ajout de condensateurs ne devrait pas être nécessaire. Cependant si les objectifs de bruit et de rebond de la tension ne sont pas atteints, il est recommandé d'ajouter des condensateurs entre 10 μF et 47 μF entre les bornes VIn et Ground et/ou VOut et Ground. Nous avons donc pris quelques condensateurs céramique multicouches CMS de 10 μF.

Afin que le patient ou les personnes assistant à la session puissent si nécessaire couper l'alimentation du système, des boutons commutateurs à bascule pour montage sur panneau ont également été commandés. Ces boutons Off-On noirs sont très petits mais pratiques d'utilisation et sont à clipser par effort de déformation dans un trou rectangulaire sur l'une des façades du boîtier d'alimentation.



(a) Convertisseur DC DC

(b) Bouton d'alimentation

FIGURE 3.8 – Composants du boîtier alimentation.

3.1.4 Conception du boîtier

A l'arrivée de Théo Molitor, étudiant en Conception et Dessin Assistée par Ordinateur, chargé de réaliser les boîtiers dans lesquels seront inclus les systèmes de stimulation, nous avons abordé le projet et notamment le choix des différents composants. A ce moment, la carte Raspberry Pi 3 B est déjà choisie, mais pas la batterie ni la majorité des composants secondaires. Nous avons donc discuté de ces composants et de leurs dimensions afin d'imaginer plus précisément l'organisation souhaitée pour les boîtiers.

Par souci pratique et pour son accessibilité, c'est Onshape qui a été utilisé pour réaliser la conception de ces boîtiers. Onshape est un système de Conception Assistée par Ordinateur entièrement en ligne, disposant d'un gestionnaire de versions et proposant un mode d'utilisation limité gratuit pour le public.

Le premier design envisagé était un boîtier unique. Afin d'optimiser le volume, une partie de ce boîtier était à deux étages, le premier contenant les têtes de réseaux Vivaltis et HiKob et le deuxième, surélevé, accueillait le contrôleur de stimulation. A coté venait se loger la batterie, dont le volume était similaire à la partie contenant le contrôleur et ses têtes de réseaux.

Cette proposition, jugée trop encombrante en un boîtier unique, a été revue en deux boîtiers distincts pouvant être placés de part et d'autre du patient. Cette séparation s'est faite avec d'un coté la batterie et de l'autre le contrôleur et ses têtes de réseaux, un câble les reliant et permettant l'alimentation du système.

Nous avons rajoutés progressivement les emplacements des divers composants en fonction de leurs dimensions, lorsque le choix s'arrêtait pour chacun d'eux, comme par exemple les embases de connecteurs Jacks en façades ou celles pour le câble d'alimentation.

Vous pouvez voir ci-dessous l'esquisse du boîtier contrôleur sur le site Onshape. Afin d'arriver aux versions finales deux versions intermédiaires ont été imprimées, grâce à l'imprimante 3D disponible au LIRMM (Laboratoire hébergeant l'équipe). Ces versions d'essai nous ont permis de vérifier les dimensions des boîtiers et de leurs logements pour les composants. Les deux images suivant l'esquisse sont les versions finales des boîtiers.

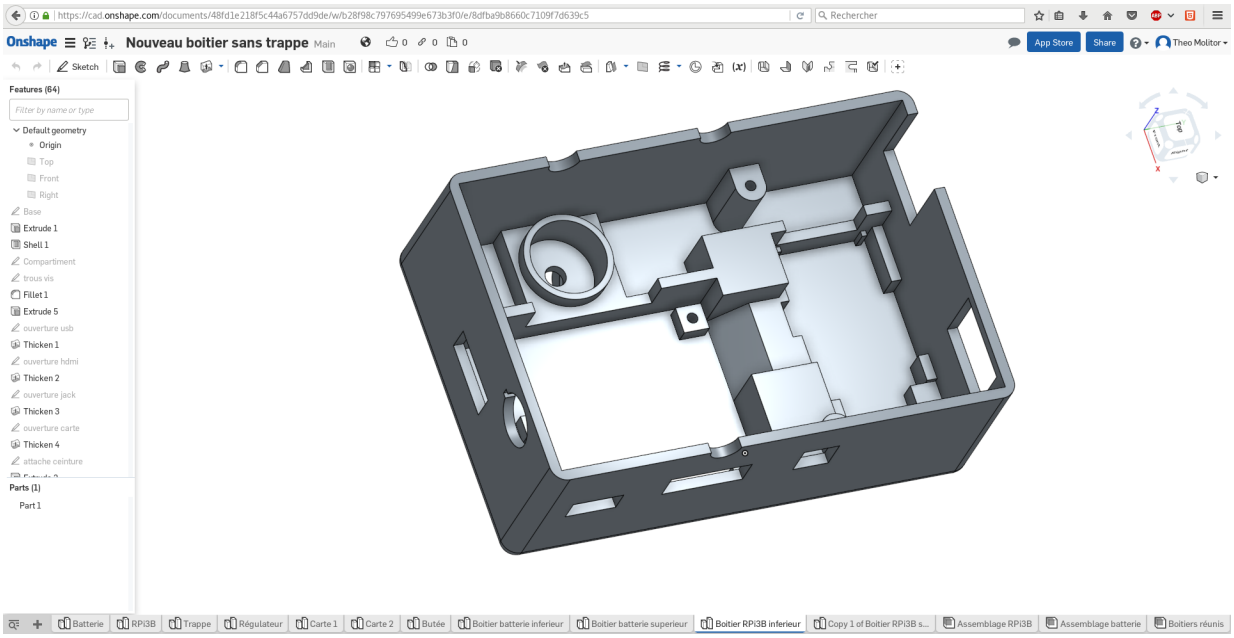


FIGURE 3.9 – Esquisse du boîtier contrôleur sur Onshape.

(a) Boîtier contrôleur contenant les têtes de réseaux et la carte Raspberry Pi.

(b) Boîtier d'alimentation contenant la batterie.

FIGURE 3.10 – Boîtiers du système final.

3.2 Partie Logicielle

Avant de développer le contrôleur, j'ai mis en place un simulateur permettant pour la suite de tester le matériel en faisant varier de façon contrôlée des paramètres du système de stimulation électrique. Afin de mettre la carte Raspberry Pi dans cet environnement il faudra modifier le programme du simulateur pour que les tâches effectuées sur le matériel soient uniquement celles qui sont sensées être réalisées dessus. Pour essayer la carte avec le simulateur il faudra d'abord la préparer et y installer un système ayant de bonnes capacités temps réel.

La boucle de contrôle qui permet d'ajuster en temps réel la stimulation électrique est la suivante :

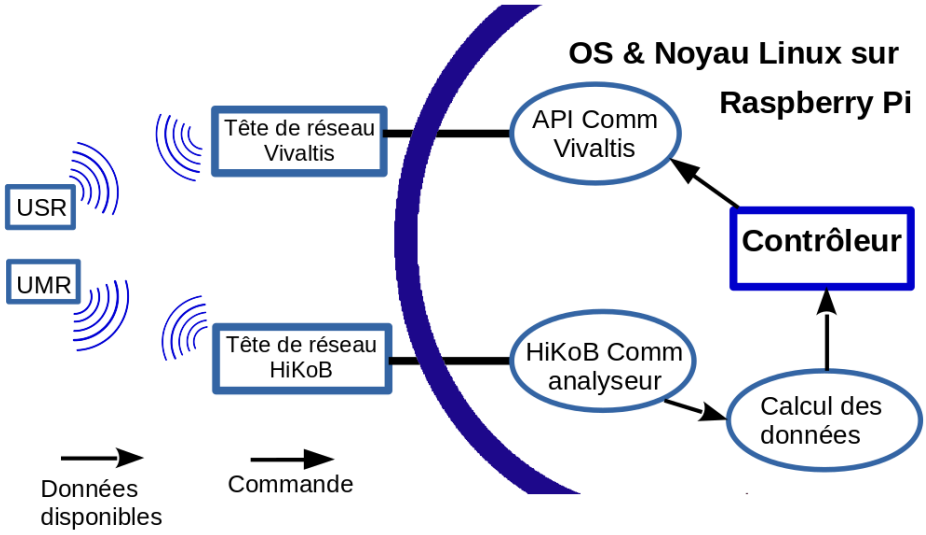


FIGURE 3.11 – Architecture logicielle principale du contrôleur de stimulation en boucle fermée.

3.2.1 Mise en place de l'OS Temps Réel

Après m'être renseigné sur les noyaux linux, leurs versions et comment les compiler manuellement, j'ai trouvé un article rédigé par Frank Dürr, un chercheur senior en informatique, détaillant une procédure de compilation croisée d'un noyau Linux pour Raspberry Pi.

L'article de Frank Dürr peut être trouvé à cette adresse :

www.frank-durr.de/ ?p=203

Les étapes sont :

— la détermination d'un noyau linux convenable, pour lequel un patch RT Preempt à été développé,
— l'acquisition du dit noyau,
— l'application du patch,
— l'installation et la configuration des outils de compilation pour le processeur fonctionnant sur la cible (Raspberry Pi : arm-bcm2708)
— la modification des options de configuration du noyau,
— la compilation du noyau, des modules et des firmwares ,
— le transfert et le remplacement du noyau et des modules/firmwares.

3.2.1.1 Compilation du noyau Linux

Avant de recevoir les cartes Raspberry Pi, je souhaitais me familiariser avec la compilation de noyau Linux et tester les performances en latence d'un système d'exploitation sous Linux, avec et sans les modifications temps réel.

J'ai donc téléchargé le noyau Linux à long terme le plus récent, le noyau 4.9.20 à ce moment, sur le site :

https ://www.kernel.org

Une version dite à long terme est une version spécifique du noyau linux, choisie pour être une base sur laquelle des projets peuvent être développés. Ces versions sont maintenues par des développeurs qui mettent en ligne régulièrement des mises à jour mineures ou de sécurité.

Après avoir décompressé les sources du noyau et installé quelques librairies et outils de développement, je suis allé modifier les options de configuration. J'ai ensuite régénéré le fichier de configuration avec la commande "make oldconfig", puis lancé la compilation du noyau avec "make bzImage". Il ne restait plus qu'à compiler les modules nécessaires allant avec ce noyau à l'aide de "make modules" et installer ces modules ainsi que le nouveau noyau via "make modules_install" et "make install".

J'ai donc pu essayer et attester du fonctionnement de ce nouveau noyau qui avait été ajouté au secteur de démarrage de mon poste de travail. Voici ci-dessous une image montrant les latences obtenues en μs en utilisant un petit programme qui effectue des cycles endormissement-réveil et détecte le décalage entre la date de réveil demandé et la date effective du réveil du programme. Cette mesure est réalisée en boucle et dans plusieurs threads en parallèle, avec différents intervalles (colonne I) entre les réveils pour chaque thread. Le script utilisé s'appelle Cyclictest. Le thread principal (numéroté "T : 0" dans la première colonne) tente ici de se réveiller toutes les millisecondes (I :1000 μs). Les deux colonnes les plus à droite représentent respectivement "Avg" l'écart moyen et "Max" l'écart maximum enregistré. Le processeur du poste de travail sur lequel est effectué ce test fonctionne à une fréquence d'horloge fixe de 2,66 GHz.

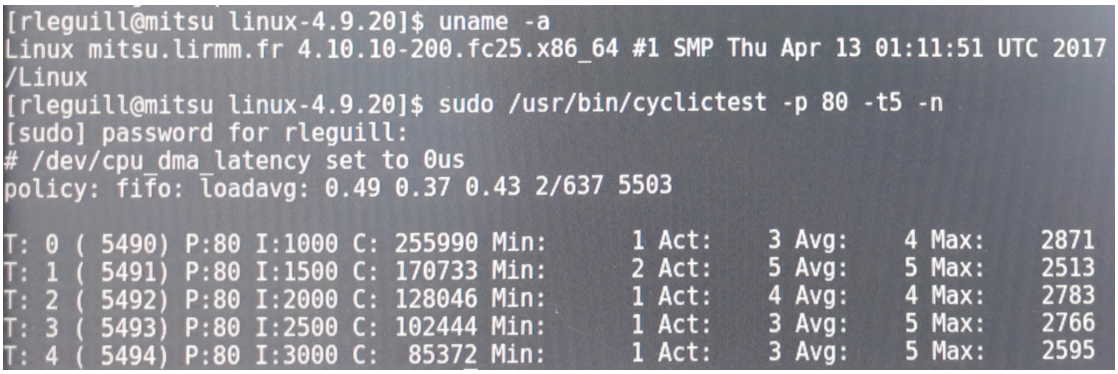


FIGURE 3.12 – Latences sur le noyau Linux sans modification

Les résultats de ce test sont intéressants au sens qu'ils indiquent qu'un système d'exploitation Linux avec un noyau sans modification a régulièrement des latences de plus d'une milliseconde. Une amélioration des latences maximales apparait nécessaire afin de satisfaire aux contraintes temps réel fermes de ce projet. Mais d'après ces résultats, il apparait malgré tout que nous ne sommes pas très loin des performances souhaitées.

3.2.1.2 Modification temps réel sur noyau Linux

Après cette première compilation j'ai recommencé l'opération mais en appliquant le patch RT_Preempt développé pour cette version du noyau Linux. On peut les trouver dans le répertoire officiel du noyau Linux à l'adresse :

https ://www.kernel.org/pub/linux/kernel/projects/rt/4.9/

Ce patch se présente sous la forme d'un fichier .patch qui représente des différences entre des fichiers et peut être utilisé via un utilitaire d'application automatique de patchs.

Dans les options de configuration du noyau il faut également faire une principale modification. Il faut activer le mode "Noyau Entièrement Préemptif (RT)" du menu "Modèle de Préemption" dans la catégorie "Fonctionnalités du Noyau". Cela permet au processeur de reprendre la main pour une opération jugée prioritaire même lorsque le noyau Linux et le système d'exploitation effectuent des tâches jugées relativement importantes. Le patch RT_Preempt applique des modifications au code du noyau pour réduire au maximum, de façon sécurisée, les sections bloquantes interdisant la préemption, c'est à dire les zones considérées critiques et n'autorisant pas le processeur à fragmenter la réalisation de la tâche en cours.

L'intérêt de cette préemption est d'éviter que le noyau n'effectue une tâche bloquante pouvant durer plusieurs millisecondes au moment même ou une réponse de l'ordre de la milliseconde ou inférieur est nécessaire. Il peut en effet arriver que toutes les tâches soient mises en pause pendant quelques millisecondes, pour une utilisation normale cela ne se remarque pas et n'a pas d'impact. Mais dans le cadre d'une application temps réel dite ferme comme dans notre cas, on souhaite éviter le plus possible que ces latences ne surviennent.

La figure 3.13 est issue du même test que le précédent, effectué en utilisant le noyau modifié et patché qui a été obtenu avec la même méthode.

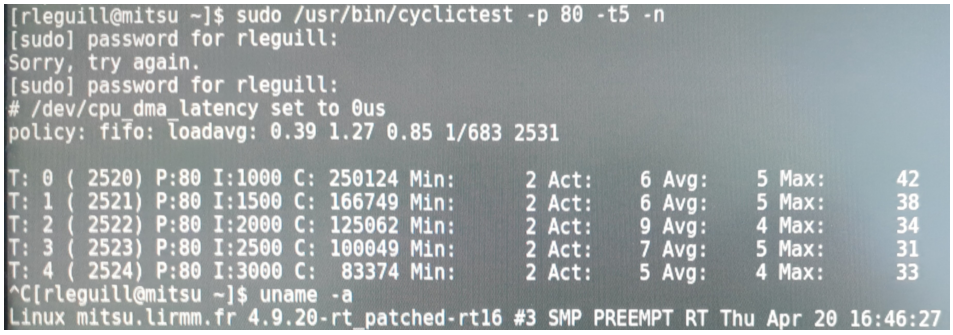


FIGURE 3.13 – Latences sur le noyau Linux patché RT_Preempt

Comme on peut le voir, sur un test équivalent nous avons une amélioration globale de la latence maximum mesurée passant de 2500μs à 40μs, soit un facteur de plus de 60. Cette amélioration s'est confirmée lorsque j'ai effectué ce test sur plus de 4 200 000 cycles. Le thread principal, le plus haut, toujours numéroté 0, effectuait cette fois-ci des réveils toutes les 100 μs.

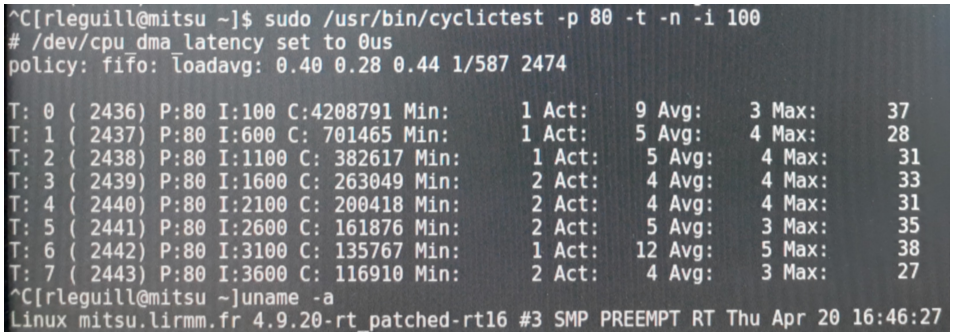


FIGURE 3.14 – Vérification des latences sur le noyau Linux patché RT_Preempt

Ces résultats sont assez satisfaisants, toutefois ils sont obtenus sur un poste de travail doté d'une grande puissance de calcul, bien plus grande que celle disponible sur la carte Raspberry Pi. De plus, le processeur de cet ordinateur fonctionne à 2,66 GHz. Ils sont donc à considérer avec précaution, mais l'amélioration qui vient d'être réalisée nous offre une marge conséquente. Ainsi il semble que nous n'aurons pas besoin de recourir à l'utilisation de solutions plus radicales et complexes, comme l'utilisation d'un co-noyau Xenomai s'occupant des tâches critiques, ou le passage à un système d'exploitation temps réel strict, aussi appelé RTOS (Real Time Operating System) comme FreeRTOS.

3.2.1.3 Compilation croisée du noyau

Afin de réaliser la compilation croisée du noyau Linux pour la carte Raspberry Pi, j'ai récupéré le projet git officiel Raspberry Pi contenant les sources du noyau Linux sur :

https ://github.com/raspberrypi/linux

La version actuelle du projet étant la même que celle du patch RT_Preempt, j'ai directement essayé d'installer le patch sur le noyau. Cependant l'application automatique du patch a rencontré quelques problèmes d'incompatibilité dans le code et j'ai du appliquer manuellement certaines modifications dans des fichiers du noyau. Une fois l'application du patch terminée, j'ai installé les outils de compilation croisée pour arm-bcm2708 puis j'ai de nouveau appliqué les modifications nécessaires aux options de configuration du noyau. J'ai également désactivé l'installation de modules inutiles, gérant la prise en charge d'une grande quantité de fonctionnalités et de matériel, afin d'alléger le noyau.

Les outils de compilation croisée pour arm-bcm2708 sont disponibles sur :

https ://github.com/raspberrypi/tools

Lors de la réception des cartes Raspberry Pi, j'ai premièrement installé une image complète du système d'exploitation officiel recommandé, Raspbian, afin de vérifier le fonctionnement de la carte et d'avoir un système prêt à installer un nouveau noyau. Ayant deux cartes micro-SD, une pour chaque kit que nous prévoyons de réaliser, j'ai cloné le système fonctionnel sur la deuxième carte afin de pouvoir recommencer facilement si un problème survenait. Après avoir compilé le noyau et ses modules sur le poste de travail pour la cible arm-bcm2708, je les ai téléchargés sur la carte Raspberry Pi et les ai finalement mis à la place du noyau et des modules existants. Le redémarrage de la carte s'est fait sans encombres et l'installation du noyau temps réel était donc terminée.

3.2.1.4 Préparation du système d'exploitation

Peu après la réception des cartes Raspberry Pi, j'ai réalisé les procédures permettant d'obtenir l'accès au réseau cablé du Lirmm ainsi qu'à son réseau Wifi. Cependant, après avoir installé le nouveau noyau modifié et les modules et firmwares appropriés, le wifi était devenu inutilisable et bien que la connexion Ethernet fonctionne, le gestionnaire de connexion sans fil était inaccessible. Ce problème précis n'apparaissait sur aucun forum d'aide pour des problèmes similaires et ce n'est qu'après maintes tentatives que j'ai résolu le problème en recompilant manuellement les firmwares de gestion de la puce wifi Broadcom 802.11 et le pilote iwlwifi du noyau Linux.

Afin de tester les performances en latences de la carte Raspberry Pi sous une grosse charge de travail j'ai réalisé un petit script lançant un navigateur internet et exécutant un test de connectivité internet, ce qui essaye de consommer le maximum de bande passante possible. Ce test est asservi en température et si elle venait à dépasser un seuil critique, le test s'arrêterait.

Le système d'exploitation avec noyau modifié était sujet à des blocages complets (freeze intégral) après quelques minutes de fonctionnement, voir quelques secondes lors d'une utilisation intensive du processeur. Ce problème n'était pas réellement documenté puisque très peu de personnes l'ont rencontré et il fut assez difficile d'en trouver la source. Ces crashs étaient apparemment dus à un conflit entre un nouveau mode de gestion rapide des interruptions USB (FIQ : fast interrupt handling) et les opérations de préemption du noyau patché. Sa désactivation sera effectivement la solution au problème de freezes du noyau.

L'image 3.15 montre le résultat de cycles réveil-endormissement de 2 millisecondes d'intervalle, effectués en continu avec le script de stress sur une durée de plus de 4 heures et demie. J'ai également préparé des scripts permettant de passer facilement le Raspberry Pi du mode performance au mode économie d'énergie dès son démarrage, ainsi que l'automatisation du lancement au démarrage de la carte d'un logiciel permettant de surveiller l'utilisation des différents cœurs du processeur et de la température de la carte. Le test suivant est réalisé en mode économie d'énergie, c'est à dire que l'horloge du processeur est ici cadencée à 600 MHz. Avec le script de stress utilisé c'est le cas le plus critique obtenu. Pour plus de tests et différents contextes, voir Annexe : "Tests de latences sur Raspberry Pi 3 B".

On observe que les résultats sont plus que corrects pour l'application désirée mais les occurrences maximales de latence sont tout de même de l'ordre de la milliseconde. Pour comparaison il est intéressant ici de noter que la fonction sleep de Python ne peut gérer des mises en attente de fonctions qu'avec une précision d'environ une milliseconde. C'est pourquoi la fonction usleep (micro sleep) n'est pas disponible en langage Python. Nous sommes donc là,

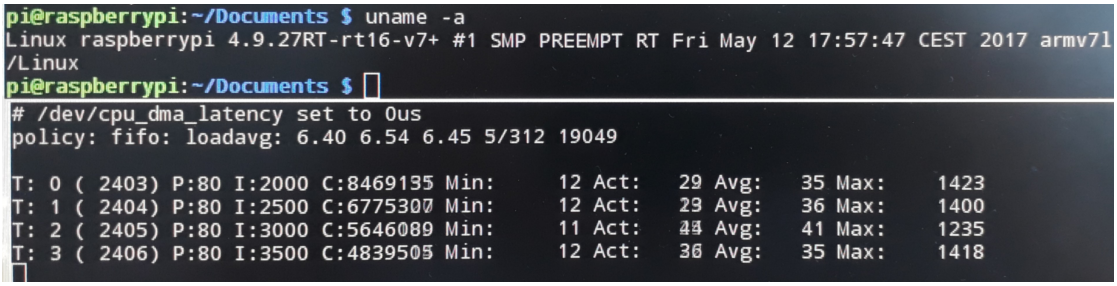


FIGURE 3.15 – Latences en utilisation intensive sur le Raspberry Pi avec noyau Linux patché RT_Preempt et horloge 600 MHz

dans les plus petites temporalités que l'on peut espérer gérer avec les objectifs et le contexte actuel du projet.

3.2.2 Mise en place du Simulateur

Le simulateur qu'il m'a été demandé de mettre en place est un projet de l'équipe CAMIN. Sa première version fut développée lors d'un stage en 2014 et a été depuis repris et approfondi principalement par mon tuteur Daniel SIMON. Ce simulateur recrée le réseau entier d'un contrôleur de stimulation et permet donc d'évaluer les performances d'un système physique en l'intégrant à un processus contrôlé, c'est à dire incorporer le matériel dans la boucle de test ("Hardware In the Loop"). Il comporte trois modules :

— **PODSTIM** : Le module PodStim émule le fonctionnement des USR (Unités de Stimulation Réparties) et des UMR (Unités de Mesures Réparties). Il contient un intégrateur numérique générant la position du genou du patient, pour les UMR, en fonction des commandes reçues par les USR.

— **CONTRO** : Le module Contro correspond au contrôleur de stimulation. Il détermine les commandes à délivrer en fonction des retours de la boucle fermée (UMR) et les transmet aux Pods de stimulation (USR).

— **MEDIUM** : Le module Medium est un intermédiaire remplissant le rôle de l'interface physique. Il redirige les messages et ajoute des délais aux communications en temps réel entre le contrôleur et les modules de stimulation et de mesures.

Ce système recrée également le protocole de communication STIMAP, utilisé pour les communications avec les stimulateurs électriques Vivaltis. Afin de faire fonctionner ce simulateur sur une plateforme quelconque, il faut y installer les APIs(Application Programming Interface) nécessaires et générer leurs librairies. Cela permet de les référencer auprès du simulateur et de faire appel à leurs fonctionnalités pour les utiliser.

Les trois APIs utilisées par le simulateur et qui sont donc à installer sont :

— **Orccad** est un IDE (Integrated Development Environment) de commandes pour des applications de robotique. Cet environnement de développement, écrit en langage C, contient entre autres une librairie de fonctions de gestion et de manipulations temps

réel POSIX (Portable Operating System Interface uniX) multi-plateformes. Elle peut être recompilée pour correspondre au fonctionnement de plateformes telles que PowerPC, Xenomai ou les systèmes courants NPTL(Native POSIX Threads Library). Elle fonctionne comme une couche d'abstraction et permet de développer des applications temps réel sur un type de système, puis si nécessaire, de les passer à un autre type de systèmes sans avoir à réaliser de modifications majeures.

— **SedSys** est une API en langage C gérant des réseaux de communications TCP et UDP, clients et serveurs, respectant la norme POSIX et étant compatible avec un fonctionnement en temps réel. Elle est utilisée pour les communications entre le contrôleur et les USR et UMR (PODSTIM), en passant également par Medium.

— **LsodaR** est un intégrateur numérique écrit en langage C, originalement développé en Fortran, permettant d'intégrer des systèmes d'équations différentielles non linéaires. Il est utilisé pour simuler en temps réel l'angle du genou du patient et permet au contrôleur d'avoir un retour physique depuis les UMR afin de recalculer et réajuster la commande de stimulation électrique.

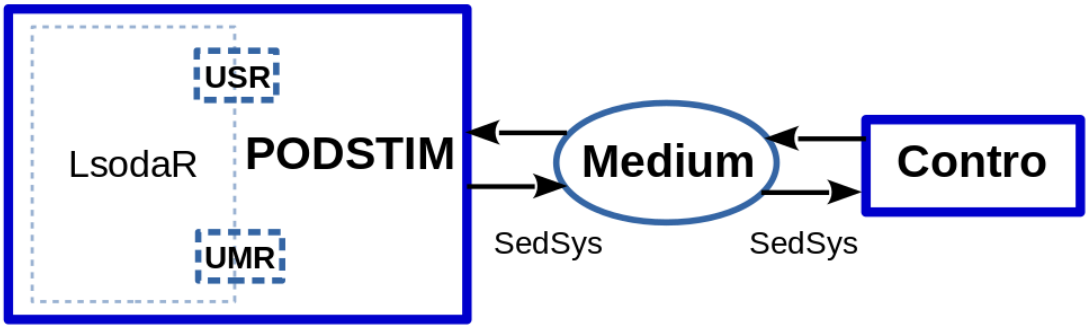


FIGURE 3.16 – Architecture du simulateur de contrôle de stimulation.

3.2.2.1 Installation du simulateur

Pour installer le simulateur il a donc fallu commencer par la compilation des librairies des trois API. Pour certaines librairies, j'ai du placer au préalable, dans le répertoire cible, un fichier librairie .a sur lequel la nouvelle librairie pourra être écrite. Après avoir compilé et généré correctement les librairies des APIs, il a été nécessaire de modifier les Makefiles des différents modules du simulateur, afin de les adapter aux nouvelles librairies et à l'arborescence de la machine. Une fois les trois modules prêts, j'ai pu lancer le simulateur en essayant différentes configurations d'initialisation grâce aux paramètres modifiables dans le fichier nommé "controlinitstate", lu au lancement du module Contro.

J'ai ensuite lu le mémoire de Master 2 du stagiaire ayant développé le simulateur pour prendre connaissance du système, puis j'ai entamé la lecture du code afin de pouvoir me l'approprier et finalement le modifier.

Après l'arrivée des cartes Raspberry Pi 3 et une fois les systèmes d'exploitation Raspbian avec noyaux modifiés temps réel en place, j'ai entrepris d'y installer le simulateur et ses modules. Après avoir compilé tous les modules et leurs librairies, j'ai tenté de lancer la simulation, sans succès. En effet, des problèmes étaient créés par le script de démarrage du simulateur. Celui-ci lance l'un après l'autre chacun des trois modules : Medium, PODSTIM puis finalement Contro. Pour laisser à chacun d'eux le temps de s'initialiser complètement avant le démarrage du module suivant, ce script utilisait la fonction usleep qui n'est pas une fonction utilisable en ligne de commande sous Raspbian.

Après avoir modifié le script pour effectuer une attente équivalente, un nouveau problème est apparu. Celui-ci était du à l'ordre d'initialisation des modules, qui se chevauchaient, au lieu d'être successifs. En effet, un retard sur le Raspberry Pi 3, par rapport aux ordinateurs habituellement utilisés, (20 centièmes de secondes environ avec le processeur travaillant à une fréquence de 1.2GHz et de 35 centièmes à 600MHz) entraînait diverses erreurs dont un "segmentation fault" de la simulation.

De plus, le calcul des nouvelles valeurs de l'intégrateur numérique, coûteux en ressources de calcul, se terminait parfois trop tard, ce qui pouvait mener à l'arrêt de la simulation. Il était donc nécessaire de découpler les modules. Le contrôleur resterait sur la carte Raspberry Pi. Le module PODSTIM, contenant les USR, les UMR et l'intégrateur numérique modélisant la position du genou, serait déporté sur le poste de travail disposant d'une plus grande capacité de calcul. Cette configuration était de toute façon l'étape suivante de la mise en place du simulateur afin d'avoir un système qui soit le plus proche possible du système réel.

3.2.2.2 Découplage du simulateur

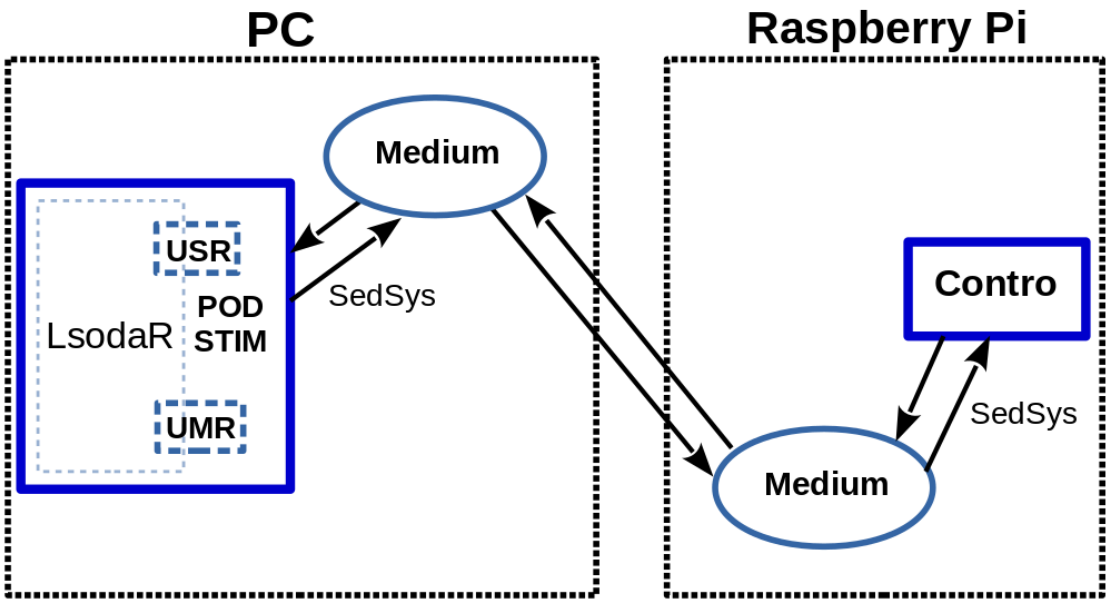


FIGURE 3.17 – Architecture du simulateur de contrôle de stimulation découplé.

La carte Raspberry Pi 3 accueille donc le module Contro, tandis que le poste de travail accueille le module PODSTIM. Cependant, chacun doit fonctionner avec un module Medium qui sert d'intermédiaire. Les modifications vont donc se focaliser principalement sur celui-ci. C'est en effet par le biais de ce module que la redirection est la plus aisée.

Afin de développer sur le même support pour les deux plateformes, j'ai créé une branche dans le projet d'origine du simulateur sur le Subversion d'INRIA forge. J'ai ensuite réalisé quelques modifications sur les Makefiles des modules, pour qu'en fonction de la présence ou non d'une variable d'environnement "Embedded" (Embarqué) sur le matériel utilisé, un paramètre soit passé lors de la compilation des modules. En fonction de ce paramètre, des parties du code seront ignorées ou prises en compte. Ce sont ces parties de codes qui seront spécifiques au Raspberry Pi, et donc au module Contro, ou au poste de travail, et donc au module PODSTIM.

Après avoir lu rapidement les fichiers sources de la librairie SedSys gérant les communications UDP et l'utilisation des sockets, j'ai réussi à appréhender plus précisément ce qui était effectué dans le code du module Medium.

Voici les étapes d'une communication entre les modules du simulateur :

1. Envoi par Contro d'une requête UDP sur le port 3500 à destination de PODSTIM.
2. Réception par Medium (com_Contro_PODS) sur le port 3500 puis envoi retardé du message en broadcast interne sur le port 9097.
3. Réception par PODSTIM du message sur le port 9097 puis envoi de la réponse sur le port 3800.
4. Réception par Medium (com_PODS_Contro) sur le port 3800 puis envoi retardé de la réponse en broadcast interne sur le port 9097.
5. Réception par Contro de la réponse sur le port 9097.
6. Envoi par Contro, si désiré, d'un nouveau message sur le port 3500 à destination de PODSTIM.
7. etc..

A l'aide de quelques modifications sur chacune des plateformes, j'ai redirigé les messages arrivant sur les ports 3500 et 3800 vers leurs destinataires respectifs en fonction du contexte. Au lieu de rediriger les messages en broadcasts internes, c'est à dire sur la même machine, le module Medium va transmettre les messages sur une adresse IP définie au préalable dans le code.

Voici les étapes d'une communication entre les modules du simulateur :

1. Envoi par Contro d'une requête UDP sur le port 3500 à destination de PODSTIM.
2. Réception par Medium_Contro (com_Contro_PODS) sur le port 3500 puis envoi retardé du message à l'adresse Computing_Hardware_IP sur le port 3500.
3. Réception par Medium_PODSTIM (com_Contro_PODS) sur le port 3500 puis envoi retardé du message en broadcast interne sur le port 9097.

4. Réception par PODSTIM du message sur le port 9097 puis envoi de la réponse sur le port 3800.

5. Réception par Medium_PODSTIM (com_PODS_Contro) sur le port 3800 puis envoi retardé de la réponse à l'adresse Embedded_Hardware_IP sur le port 3800.

6. Réception par Medium_Contro (com_PODS_Contro) sur le port 3800 puis envoi retardé de la réponse en broadcast interne sur le port 9097.

7. Réception par Contro de la réponse sur le port 9097.

8. Envoi par Contro, si désiré, d'un nouveau message sur le port 3500 à destination de PODSTIM.

9. etc..

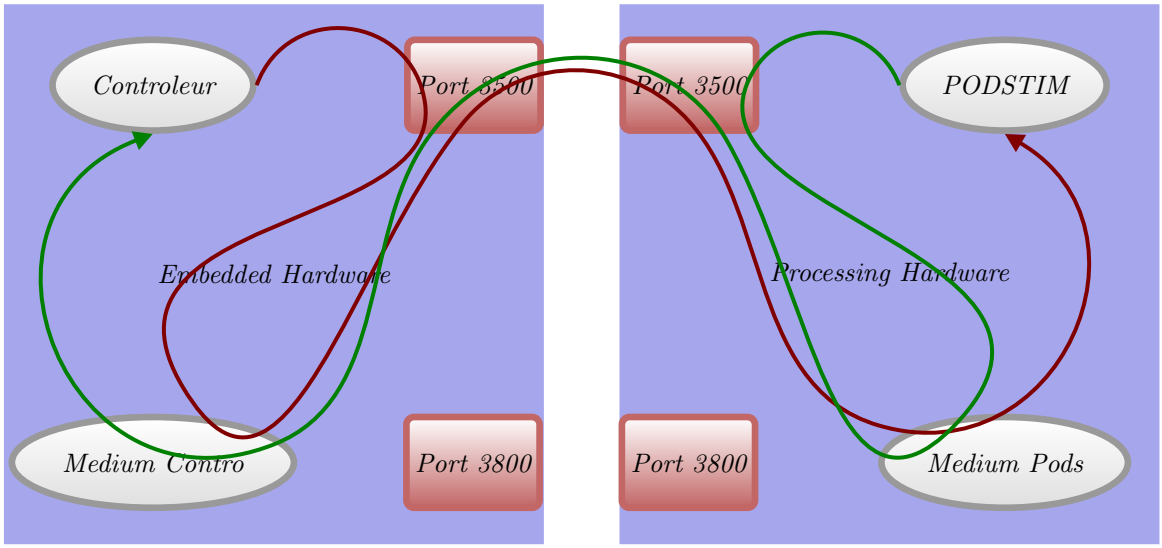


FIGURE 3.18 – Contro envoyant un message à PODSTIM en rouge, via le port 3500 et PODSTIM répondant à Contro en vert

Après avoir mis cela en place entre les deux plateformes, en démarrant les modules dans l'ordre, les modules Medium d'abord puis PODSTIM et finalement Contro, la simulation démarrait mais s'arrêtait généralement après quelques échanges de trames. Le premier message d'erreur apparaissant était "Out of range which is 7 -> this -> size() 3", ce qui indique une sortie de l'espace mémoire attribué. Afin de trouver la source ce problème j'ai analysé les situations ou des comparaisons directes ou "statiques" étaient effectuées entre des buffers d'octets de données, des tableaux de valeurs ou des chaines de caractères. J'ai appliqué diverses modifications dans le but de résoudre le problème, sans succès.

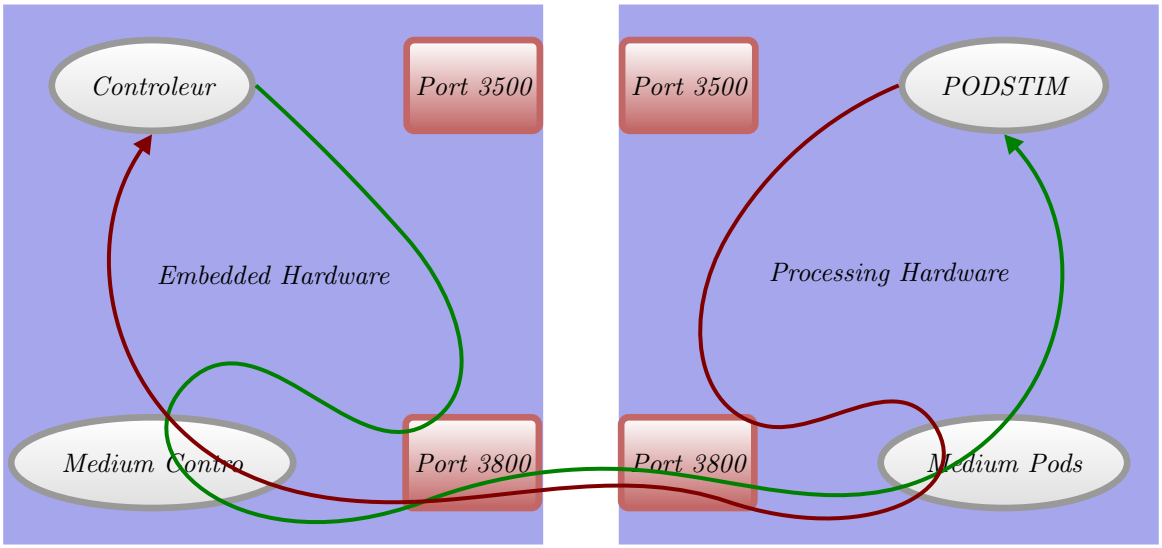


FIGURE 3.19 – PODSTIM envoyant un message à Contro en rouge, via le port 3800 et Contro répondant à PODSTIM en vert

Je me suis ensuite rendu compte que les tailles en octets des variables étaient différentes entre les architectures 64bits et 32bits du poste de travail et de la carte Raspberry Pi. En effet, les types "long int" par exemple, étaient de 4 octets sur la carte Raspberry Pi 3 B et de 8 octets sur le poste de travail. Ce qui pouvait entrainer des problèmes à n'importe quel endroit ou des types contenant le préfixe "long" étaient utilisés. Cependant, la majorité du code des modules et des librairies utilisaient des types redéfinis, explicités et donc communs sur différentes plateformes.

```
pi@raspberrypi:~/Documents $ uname -a
Linux raspberrypi 4.9.27RT-rt16-v7+ #1 SMP P
pi@raspberrypi:~/Documents $ ./testarch
Taille des types (sizeof()):
* char:          1
* short:         2
* int:           4
* unsigned int:          4
* long unsigned int:             4
* long long unsigned int:                8
* long int:      4
* long long:     8
* float:         4
* double:        8
* long double:   8
```

```
[rleguill@mitsu Downloads]$ uname -a
Linux mitsu.lirmm.fr 4.10.13-200.fc25.x86
[rleguill@mitsu Downloads]$ ./testarch
Taille des types (sizeof()):
* char:          1
* short:         2
* int:           4
* unsigned int:          4
* long unsigned int:             8
* long long unsigned int:                8
* long int:      8
* long long:     8
* float:         4
* double:        8
* long double:   16
```

FIGURE 3.20 – Tailles en octets des types C sur architectures 32bits Raspberry Pi 3 B à gauche et 64bits à droite sur le poste de travail

Ne trouvant pas d'options de compilation permettant d'expliciter au compilateur gcc les tailles des types à utiliser, j'ai donc entrepris de changer systématiquement dans le code du simulateur, les types "long int" par des types "long long int". Le type "long long int" étant de 8

octets sur les deux plateformes. Cependant, les librairies telles que SedSys et Orccad utilisaient elles aussi parfois ces types supportés de manières différentes sur les deux plateformes. Et il allait être compliqué de les modifier de la même façon car les dépendances étaient plus complexes.

Ne voyant pas de solutions directes, je suis retourné à une version non modifiée du simulateur pour éviter que des modifications ultérieures ne puissent venir perturber le programme et interférer avec la résolution du problème. Puis en vérifiant à nouveau les opérations effectuées sur les trames de communications, j'ai découvert la source du problème. Le résoudre n'a finalement nécessité qu'une modification légère du fichier POD.c du module PODSTIM. (Révision numéro 148 de la branche stimpi)

En effet, après la tentative de lecture d'un message de la part du module Contro, les différentes possibilités étaient vérifiées. Ces possibilités incluent également l'absence de communication dans le temps imparti, c'est à dire le "Time Out" (ou "temps écoulé"). La première possibilité vérifiée était une trame correspondant à la demande d'arrêt des Pods de stimulation. Cependant, si aucune réponse n'était arrivée, le buffer de lecture ne contiendrait que les trois caractères suivants, écrits par défaut : "TOP" (Time Out Pod). Lors d'un Time Out, afin d'éviter l'accès à un espace mémoire non alloué en comparant de façon statique le buffer de lecture avec la trame STOP, il était donc nécessaire de traiter en premier lieu la possibilité "Time Out Pod".

Après cette modification le simulateur découplé était fonctionnel et j'ai donc pu réaliser quelques tests sur des durées variées et avec divers paramètres d'initialisation. Il pourra plus tard être utilisé pour évaluer les performances de la partie matérielle du contrôleur dans un environnement maîtrisé et servir de base pour le développement d'un contrôleur de Stimulation Électrique Fonctionnelle en langage C++. Le simulateur maintenant en place, je pouvais passer au développement du contrôleur de stimulation.

3.2.3 Développement du contrôleur

L'architecture logicielle du contrôleur est composée de trois parties principales. La première est l'interface entre le contrôleur de stimulation et le réseau HiKoB qui est effectuée grâce à l'analyseur de trames HiKoB. La deuxième est l'interface entre le contrôleur et le réseau Vivaltis, fournie cette fois-ci par une librairie de fonctions que le contrôleur peut utiliser afin de communiquer avec le réseau ou d'envoyer des commandes. La troisième partie correspond au module fermant la boucle et permettant au contrôleur d'effectuer des vérifications ou des calculs divers sur les données obtenues grâce au réseau de capteurs HiKoB. Le contrôleur peut ainsi réévaluer la situation et modifier la stimulation électrique si besoin. Une fonctionnalité importante du contrôleur de stimulation est la gestion des interruptions provenant de l'extérieur du système, en effet elle permet de réaliser des mesures au déclenchement matériel d'un événement ou de prendre en compte diverses informations supplémentaires.

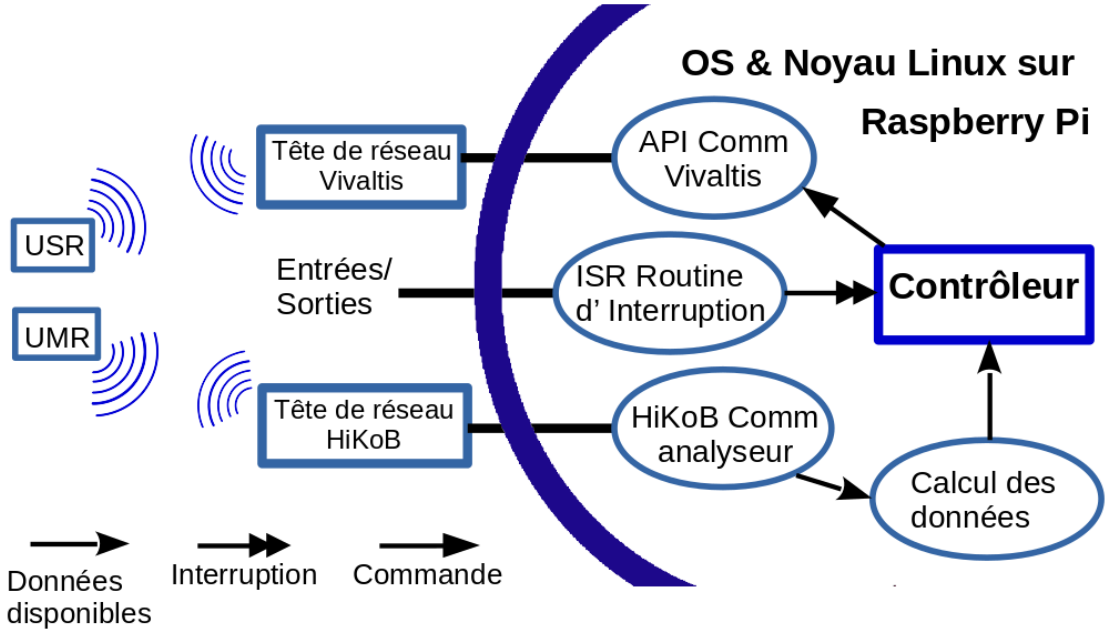


FIGURE 3.21 – Architecture logicielle du contrôleur.

Après la mise en place du simulateur, nous avons fait une réunion avec mes tuteurs afin de discuter des étapes de développement du contrôleur de stimulation embarqué et de classer par ordre de priorité les tâches principales du projet.

Les objectifs définis lors de cette réunion étaient (par priorité décroissante) :

1. Communiquer avec les réseaux HiKoB (UMR) et Vivaltis (USR). C'est à dire récupérer les données des centrales inertielles et contrôler la stimulation.
2. Obtenir la métrologie du système pour pouvoir réaliser un ordonnancement régulé des communications.

3. Gérer l'enregistrement des données et leur transfert sur l'ordinateur de contrôle.
4. Mettre en place la prise en charge de boutons externes, par traitement d'interruptions si possible ou par scrutation.
5. Avoir une interface graphique permettant de moduler la stimulation ou de choisir des profils prédéfinis.

J'ai ensuite échangé avec le doctorant, Benoît Sijobert, principal utilisateur potentiel du système, au sujet des librairies Python utilisées pour communiquer et recevoir les données des capteurs. Ce sont ces données qui permettent d'avoir un retour sur la stimulation et de réaliser une boucle fermée. Il m'a fourni les scripts Python et Matlab qu'il utilisait pour ses expérimentations, ainsi que l'API de communication Vivaltis (Unités de Stimulation Réparties) en Matlab et le script d'analyse des trames de communication HiKoB (Unités de Mesures Réparties). J'ai donc commencé par lire ces scripts afin de les prendre en main. Suite à la mise en lumière de complications et de limitations lors de l'utilisation de librairies Python via un processus écrit en langage C++/C, nous avons décidé de développer le contrôleur premièrement en langage Python. Afin de recueillir les besoins et usages, le développement du contrôleur s'est fait par la suite en étroite collaboration avec Benoît Sijobert.

3.2.3.1 Développement de l'API Vivaltis

Après avoir pris connaissance des fonctions de communication avec la passerelle Vivaltis en langage Matlab, j'ai commencé à les traduire en Python. J'ai progressivement remarqué les subtilités de certaines conversions automatiques de types dans divers contextes et en particulier lors de communications octet par octet sur un port série. En effet, pour que les trames de communication soient conformes au protocole Vivaltis, les variables sont chacune limitées à un nombre précis d'octets. J'ai donc ensuite créé une petite librairie de fonctions permettant d'effectuer des opérations équivalentes en Python. Elle permet par exemple de transformer des entiers, des nombres hexadécimaux ou des chaines de caractères correspondant à des nombres hexadécimaux, en une suite d'octets selon la longueur souhaitée, que l'on peut ensuite envoyer sur la liaison série. J'ai également implémenté des structures d'objets en Python, équivalentes à celles utilisées en Matlab afin de faciliter la traduction. Je me suis plus tard écarté, pour simplifier le code, de l'utilisation de certaines de ces structures après avoir mieux compris le fonctionnement du protocole de communication.

J'ai commencé les tests avec une écriture simple dans un fichier, puis j'ai utilisé une tête de réseau et un Pod Vivaltis afin de mettre en place le fonctionnement. Après divers tests et la lecture d'une version non exhaustive du protocole de communication, j'ai finalisé la communication avec la passerelle Vivaltis et pu visualiser la stimulation électrique délivrée sur un oscilloscope.

La mise en place de cette API sur le Raspberry Pi a entrainé l'apparition de plusieurs problèmes dus au portage ainsi qu'aux différentes versions des librairies et fonctions disponibles. Afin de déterminer le moyen le plus sûr pour envoyer les trames octet par octet en Python, et la façon de les lire en retour, je me suis familiarisé avec le fonctionnement de la librairie Py Serial. L'idéal était apparemment d'envoyer la trame en la convertissant au préalable avec la fonction serial.to_bytes() puis de récupérer les octets en les lisant comme des caractères Unicodes avec ord(). J'ai pu ensuite expérimenter avec la stimulation modulée en largeur d'impulsion par le Raspberry Pi.

Les fonctions disponibles étaient alors :

— Ouvrir la liaison série correspondant à la passerelle Vivaltis et initialiser le réseau.
— Initialiser un Pod Vivaltis pour le type de stimulation souhaité.
— Modifier la largeur d'impulsion d'un Pod.
— Signaler à un Pod d'arrêter toute stimulation.
— Fermer la liaison série avec la passerelle Vivaltis.

3.2.3.2 Contrôleur minimaliste de stimulation

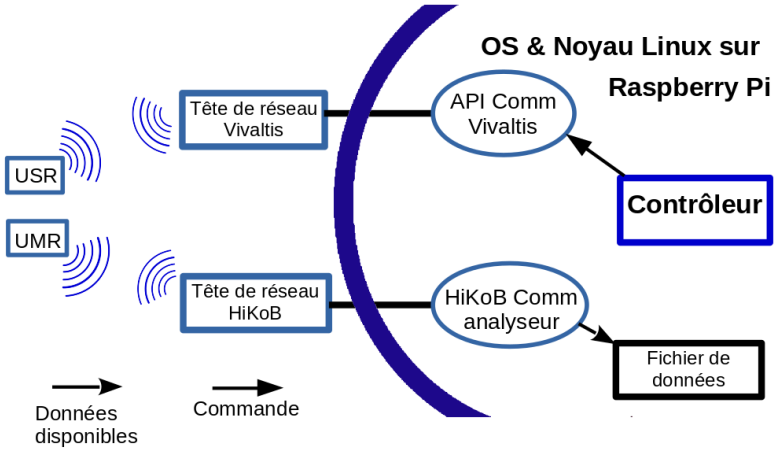


FIGURE 3.22 – Architecture logicielle de l'ébauche de contrôleur.

Après avoir développé l'API de communication pour le réseau Vivaltis, j'ai pu commencer à créer un programme principal supervisant la stimulation et commencer à lui ajouter quelques fonctionnalités. L'un des objectifs étant de gérer des interventions externes, j'ai mis en place un bouton poussoir géré par interruptions via les GPIO du Rasperry Pi. A celui-ci j'ai associé une Routine de Service d' Interruption (ISR) traitant l'interruption matérielle et modifiant la stimulation en incrémentant la largeur l'impulsion. J'ai ensuite développé une Interface Graphique Utilisateur (GUI) minimaliste afin de permettre à l'utilisateur de prendre le contrôle sur le programme pendant son fonctionnement et modifier les paramètres de stimulation. En autorisant le déport d'affichage (X11 Forwarding) via le protocole d'accès à distance SSH, j'ai pu essayer avec succès le lancement à distance, avec interface graphique, du contrôleur de stimulation sur le Raspberry Pi, depuis le poste de travail.

Dans l'objectif d'obtenir une gestion dynamique et automatique des nœuds HiKoB présents sur le réseau, j'ai réécrit une grande partie du script Python analysant les données transmises par la tête de réseau HiKoB. Au lieu de supposer un nombre précis de centrales inertielles, celui-ci pourra maintenant être défini au lancement du programme. J'ai également intégré, sur demande de Benoît Sijobert, un système détectant les prises de paroles entre les nœuds et déclenchant des avertissements ou même une alerte mettant fin à la stimulation si un nœud n'a pas "parlé" depuis plus longtemps que la limite définie au préalable. J'ai ensuite mis en place la lecture et

le traitement des données transmises depuis les centrales inertielles dans un processus autonome créé au démarrage du contrôleur.

3.2.3.3 Contrôleur pour détection de périodes non-stationnaires

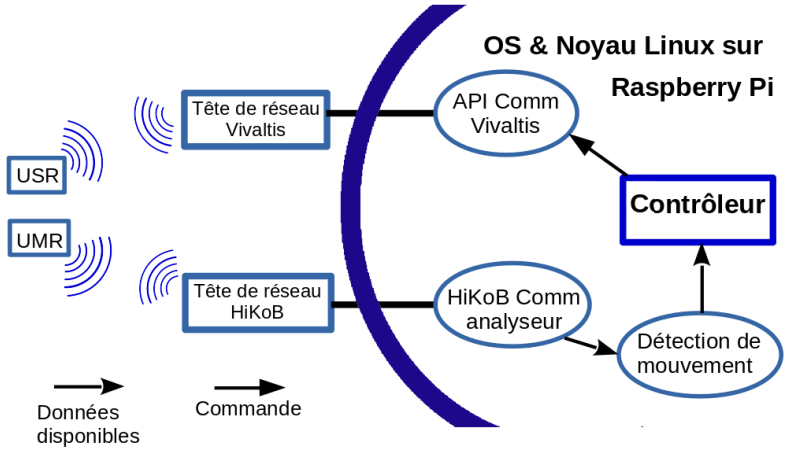


FIGURE 3.23 – Architecture logicielle du contrôleur détectant les périodes non-stationnaires.

Afin de développer et tester plus amplement le système il me fallait maintenant mettre en place une application réelle. Benoît Sijobert m'a donc donné un script Python, Parkinson_cueing, permettant de détecter des tremblements via les centrales inertielles. L'objectif étant de déclencher la stimulation électrique et donc de contracter les muscles, si un tremblement est détecté chez le patient.

J'ai modifié le script pour l'adapter au fonctionnement avec le contrôleur et l'ai incorporé à un nouveau processus autonome, créé par le contrôleur à son initialisation. Il a fallu ensuite modifier la gestion des données transmises par la tête de réseau HiKoB, auparavant écrites dans un fichier à la réception de la trame de d'informations. Les données sont maintenant stockées dans un tableau dont la taille est définie par le nombre de nœuds HiKoB sur le réseau. Ce tableau, muni de sécurités le protégeant d'accès simultanés, est partagé entre le processus analyseur des trames HiKoB et le processus de détection de périodes non-stationnaires. Des drapeaux également sécurisés en accès inter-processus permettent de faire remonter les informations rapidement d'un processus à l'autre, et permet au contrôleur de délivrer la stimulation électrique si un mouvement est détecté.

Afin d'estimer les performances du contrôleur, j'ai ensuite implémenté un système de sauvegarde des dates d'exécutions ("time-stamps") dans des variables à accès protégé, permettant de déterminer les diverses durées au cours du processus entier. Mes premiers essais présentaient des résultats étranges de performances excellentes lorsqu'une durée maximum d'attente de la réponse à une requête ("time out") était appliquée. Après avoir remarqué qu'il était nécessaire de vider ("Flush") la communication série de tout octet présent avant d'envoyer une requête, pour éviter que la réponse du cycle précédent n'interfère, j'ai obtenu des résultats plus cohérents. Cela m'a permis d'estimer la durée du cycle du contrôleur : réception des

données HiKoB, traitement de ces données et application effective de la stimulation à environ 30 millisecondes en moyenne. Soit une fréquence de rafraichissement que l'on peut supposer d' approximativement 30 Hz.

Voici ci-dessous un histogramme répertoriant les occurrences des durées de ces cycles. En haut à gauche se trouvent les durées entre la réception des nouvelles données de la centrale inertielle que l'on souhaite analyser, et le calcul de ces données pour déterminer si ce noeud HiKoB est dans une période stationnaire ou non-stationnaire. En haut à droite se trouvent les durées entre la détection ou non d'une période stationnaire et l'application de la stimulation en conséquence. Le dernier histogramme est la somme de ces deux durées et correspond donc aux durées des cycles. Dans ce cas, la durée maximum passée à attendre la réponse à une requête de modification de la largeur d'impulsion de la stimulation est de 15 millisecondes. Divers tests similaires dont un avec un "time out" d'une seconde, c'est à dire en pratique une attente systématique des réponses, ont globalement confirmé cette répartition. Toutes les durées des histogrammes sont en μs.

J'ai pu également déterminer, en les testant de façon spécifique, les temps de propagation de modifications des variables et des événements partagés par des processus, sur le Raspberry Pi fonctionnant à 1,2 GHz. D'après mes observations ces temps de propagation se situent à 400 μs pour les variables (utilisées ici comme des drapeaux) et 800 μs pour les événements.

L'un des objectifs principaux du projet était de permettre l'accès et le contrôle à distance du contrôleur de stimulation, de préférence sans nécessiter un accès à internet. En effet avec ce fonctionnement, un clinicien peut superviser la situation avec précision pendant que le patient déambule. J'ai donc étudié le fonctionnement des communications particulier-à-particulier(peer-to-peer), mode de communication sans fil qui semblait être supporté par le Raspberry Pi 3 B. Après plusieurs essais infructueux et quelques redémarrages du système, j'ai finalement réussi à configurer la carte de communication sans-fil du Raspberry Pi sur un canal spécifique de la bande de fréquences 2.4GHz. J'ai pu ensuite m'y connecter avec un autre ordinateur sous Linux possédant une carte Wifi, en entrant les configurations en lignes de commandes sur les deux plateformes en même temps. Pour simplifier ce fonctionnement et permettre à un utilisateur de se connecter au Raspberry Pi directement à son démarrage sans accès à internet, j'ai configuré la carte Rapsberry pour qu'elle fonctionne en "Hot-Spot". Elle crée donc automatiquement un réseau détectable par d'autres plateformes et sur lequel elles peuvent se connecter en s'attribuant une adresse IP statique.

Un autre objectif principal était de transmettre l'enregistrement des informations acquises pendant la session de stimulation ("Récupération des logs"). Grâce à ces enregistrements, il sera possible même après les protocoles cliniques de retracer les graphes de données. J'ai donc ajouté un fonctionnement automatique à la fin de la session qui rapatrie un fichier contenant les données accumulées sur l'ordinateur de contrôle via le protocole SCP, basé lui même sur le protocole SSH. Pour cette version, le nom de ce fichier devra être donné par l'utilisateur via l'interface graphique avant de pouvoir démarrer la session.

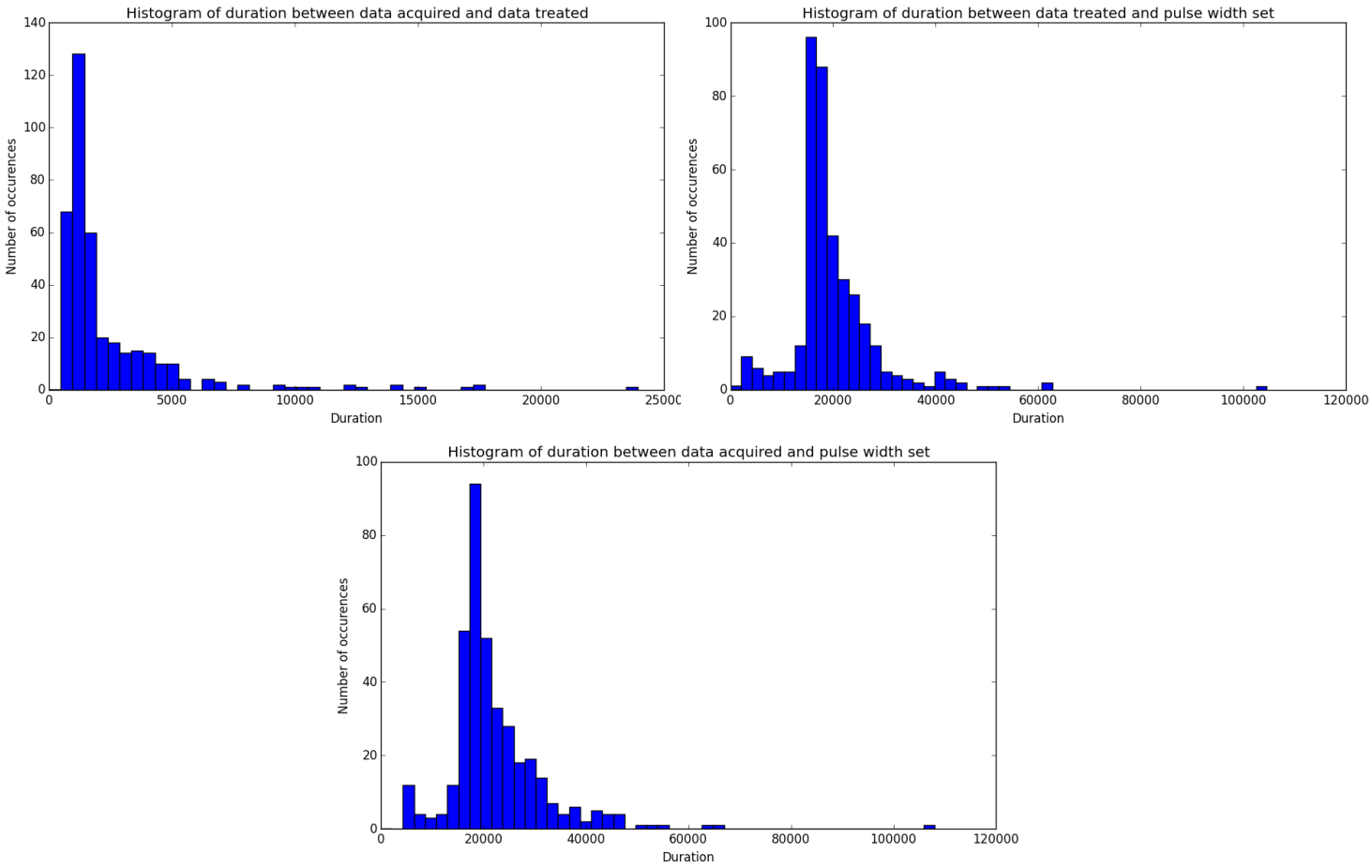


FIGURE 3.24 – Durées des parties Réception-Traitement et Traitement-Stimulation du cycle de contrôle ainsi que les durées du cycle global pour Parkinson Cueing.

3.2.3.4 Contrôleur pour une application de marche

Par la suite j'ai effectué quelques tests supplémentaires de la configuration "Hot-Spot" de la carte Raspberry Pi et de son fonctionnement. Afin d'éviter qu'une personne quelconque puisse aisément interférer avec le système en fonctionnement, j'ai ajouté l'utilisation d'une clef WEP statique, offrant un minimum de sécurité. Malheureusement, les clef WPA, utilisant un mécanisme de chiffrement dynamique plus sûr, ne sont pas supportées par le gestionnaire Wifi WPA_supplicant de la carte Raspberry Pi en mode Ad-Hoc. J'ai également essayé le lancement du contrôleur, avec interface graphique, depuis des ordinateurs Windows. Pour y arriver il faut utiliser conjointement les logiciels Putty (Emulateur de terminal intégrant un client pour divers protocoles comme SSH, Telnet et TCP) et XMing(Système de fenêtrage graphique X).

Une réunion d'avancement a permis de définir des optimisations nécessaires du code ainsi que l'ajout de quelques fonctionnalités. J'ai ensuite modifié dans le programme la majorité des fonctionnements par drapeaux et des boucles d'actualisation utilisés pour le développement, au

profit d'un fonctionnement évènementiel dans le but de gagner en performances globales et en utilisation de temps de calcul.

Afin de mettre à l'échelle le système, c'est à dire ici de le mettre en condition réelle, avec 4 nœuds HiKoB sur le réseau et devant contrôler plusieurs canaux de stimulation, je suis passé à la mise en place d'une application de marche. Les Unités de Mesures Réparties seront placées sur les jambes du patient, autour de ses genoux, l'un au-dessus sur la cuisse et l'autre en dessous sur le mollet. Cette application doit récupérer les données des deux duos de centrales inertielles, déterminer les angles présents entre les nœuds HiKoB de chaque couple et appliquer en conséquence la stimulation sur les Pods Vivaltis en temps réel. Afin de visualiser aisément le fonctionnement, ces stimulations seront des représentations des angles formés par les genoux du patient.

Le script comportant les fonctions permettant d'effectuer les calculs d'angles grâce aux données des accéléromètres et des gyroscopes des centrales inertielles a été assez rapide à intégrer au contrôleur. J'ai donc développé une nouvelle version du contrôleur, intégrant à la place du processus Parkinson_cueing le processus goniomètre temps réel pour 4 nœuds. Il est intéressant de noter que la présence de 4 centrales inertielles sur le réseau HiKoB résulte en une augmentation significative des inégalités de temps de parole entre eux. Il peut arriver qu'un nœud ne parle pas pendant plus d'une vingtaine d'actualisations. Les seuils d'avertissements et d'alertes du système ont dû être assouplis en conséquence.

La commande successive de 2 canaux étant lente (2 fois 20ms au minimum environ), j'ai tenté de réduire le temps passé à attendre les réponses ("acknowledgement") par l'utilisation de threads s'occupant des fonctions de communication. Sans succès, car la communication série avec la tête de réseau Vivaltis semble rendre impossible un traitement parallèle ou quasiment parallèle des requêtes. J'ai entrepris après cela de simplifier et modifier un peu le fonctionnement du programme.

J'ai encapsulé l'envoi des trames de communication Vivaltis dans une nouvelle fonction de l'API, cette fonction générique prend en entrée la trame à envoyer et une valeur de "time out" à utiliser. Cette modification permet de gérer un "time out" dynamique, variant entre les envois de trames. J'ai ensuite utilisé cette fonctionnalité pour faire évoluer la valeur de "time out" afin d'atteindre un taux de réussite prédéfini d'acquittement des requêtes. En effet, au delà d'un certain seuil, il est préférable pour ce type d'application d'abandonner la finalisation de l'envoi d'une requête si celle-ci prend trop de temps, afin d'éviter de perdre en efficacité, de bloquer le système voire de mettre en danger le patient.

Si trop de modifications de la largeur d'impulsion des pods de stimulation n'arrivent pas à recevoir leur acquittement dans le temps imparti, cette durée ("time out") sera augmentée en conséquence. Inversement, s'il est possible de réduire la valeur d'attente maximum tout en restant dans le taux de réussite désiré, celle-ci sera réduite afin d'optimiser les attentes et d'améliorer les performances du système.

Voici un histogramme, similaire à celui présenté précédemment, montrant les performances du système utilisant ce fonctionnement de "time out" dynamique. Le contrôleur envoie, dans cet exemple, systématiquement et de façon séquentielle, les trames de modification de largeur d'impulsion sur les deux canaux de stimulation d'un Pod Vivaltis. Les durées présentées dans cet histogramme sont en μs.

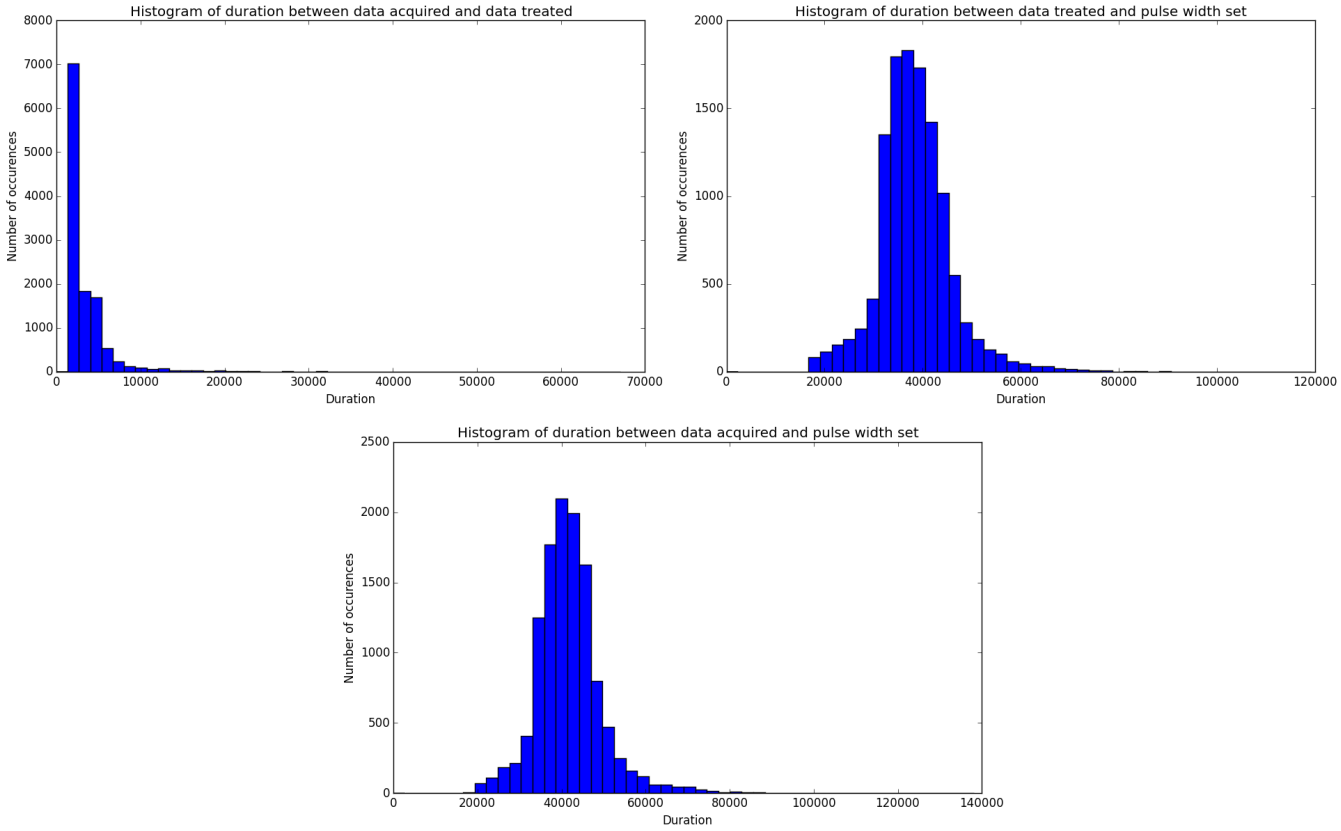


FIGURE 3.25 – Durées des parties Réception-Traitement et Traitement-Stimulation du cycle de contrôle ainsi que les durées du cycle global pour le goniomètre.

Après avoir relu le protocole de communication Vivaltis et trouvé la structure des trames permettant de demander le niveau des batteries des Pods ainsi que d'incrémenter ou décrémenter l'intensité de la stimulation, j'ai ajouté ces fonctionnalités au contrôleur. Il a été aisé de les implémenter grâce à la nouvelle fonction gérant l'envoi des requêtes et la réception de leurs réponses. Afin de pouvoir utiliser ces fonctionnalités j'ai rajouté à l'interface graphique de l'utilisateur les boutons et affichages correspondant. Pour calibrer les centrales inertielles et choisir le référentiel des angles des genoux j'ai ajouté sur l'interface la possibilité de recalculer l'offset utilisé dans le calcul des angles. On peut donc à présent incrémenter ou décrémenter l'intensité des impulsions de stimulations sur les canaux utilisés, actualiser l'affichage du niveau des batteries des Pods et re-calibrer les angles des nœuds HiKoB. Cela m'a d'ailleurs permis de remarquer une dérive progressive des angles calculés grâce aux données HiKoB. Sans qu'il n'y ait de mouvement, l'angle calculé peut dériver jusqu'à plus 0.3 degré par seconde.

Pour finir j'ai ajouté un système d'optimisation des envois de trames d'actualisation des largeurs d'impulsions afin éviter de surcharger le réseau si aucune modification significative n'est à appliquer. Si seul un angle vient d'être actualisé par exemple, on enverra uniquement la trame permettant de modifier la largeur d'impulsion correspondante. Cela permet d'augmenter

significativement les performances du système lors de périodes sans mouvement ou de mouvements alternatifs.

Voici un nouvel histogramme montrant les délais entre la réception et le traitement des données des centrales inertielles, en haut à gauche, le traitement des données et la stimulation conséquente en haut à droite, ainsi que la somme de ces deux durées dans l'histogramme du bas.

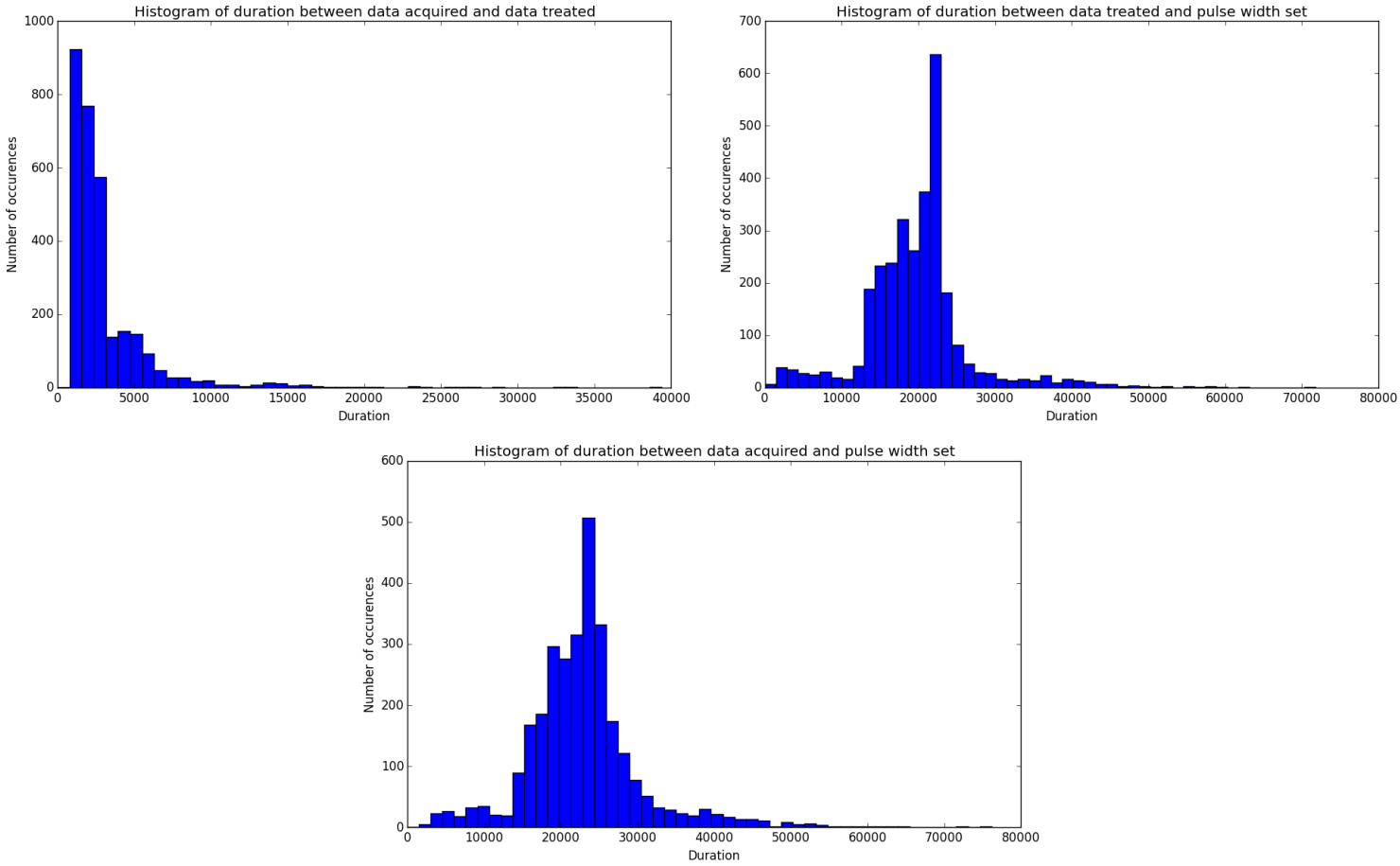


FIGURE 3.26 – Durées des parties Réception-Traitement et Traitement-Stimulation du cycle de contrôle ainsi que les durées du cycle global pour le goniomètre optimisé.

3.2.3.5 Réalisation du test de latence globale du système

Afin de déterminer la réactivité réelle du contrôleur dans sa globalité, Benoît Sijobert et moi avons mis en place un système de test intégrant une carte d'acquisition pouvant démarrer la mesure lors du déclenchement d'une interruption matérielle ou du passage des centrales inertielles à un angle précis. En prévision de ces tests j'ai ajouté des mesures dans mon programme afin d'estimer d'abord logiciellement la latence entre deux actualisations de la stimulation. Latence que j'ai pu estimer à 50 millisecondes en moyenne et à un maximum d'environ 60 millisecondes pour couvrir 90% des cas. J'ai également mis en place de nouveaux fichiers de "logs" permettant de sauvegarder les résultats et de visualiser les angles des genoux.

Après que Benoît Sijobert ait préparé le système d'acquisition, nous avons installé le contrôleur. Dès les premiers essais concluants, nous nous sommes rendus compte que la réactivité du système dans sa globalité sur ce test était bien moins bonne que ce que nous pensions. En effet la réactivé semblait être de l'ordre de 200 millisecondes. Les nœuds HiKoB introduisent une latence d'angle assez forte, allant jusqu'à 20 degrés de retard sur un mouvement rapide, ce qui peut expliquer une partie conséquente de ce retard étant donné les critères de ce test. Nous n'avons pas pu investiguer plus avant sur ce point car ces essais ont mis également en évidence un autre problème. Après être remonté progressivement à la source de celui-ci, il apparait qu'il y a un problème d'incompatibilité relative avec la connexion USB de la tête de réseau Vivaltis. En effet, le driver pour la connexion USB de la tête de réseau Vivaltis est sensé être un driver FTDI D2XX et non un FTDI VCP. Sachant que la tête de réseau HiKoB elle, doit être utilisé via un driver FTDI VCP, il n'est pas sûr qu'il y ait une solution directement applicable. Cependant le système d'exploitation du Raspberry Pi semble réussir à contourner le souci et la communication fonctionne donc mais utilise une quantité considérable de temps de calcul et est potentiellement instable lors d'une utilisation en continu. Malheureusement, ce problème, détecté à la fin de mon stage, risque de devoir être résolu plus tard afin de rendre le système exploitable en situation réelle.

Nous avons aussi découvert que l'utilisation d'un "time out" dans l'attente d'un évènement en python via la librairie "multi processing" était basée sur l'utilisation d'endormissements ("sleeps") successifs et introduisait donc une latence de plusieurs millisecondes dans la détection potentielle de l'évènement. Ce point, très peu documenté sur internet complique sensiblement l'utilisation d'un système basé sur un fonctionnement évènementiel en Python.

Pour ce qui est de la réactivité du contrôleur en lui même, en revanche, j'ai pu l'estimer en déclenchant la stimulation électrique sur un Pod Vivaltis à la détection d'une interruption matérielle. Ce test a été réalisé en laissant tourner de façon autonome le processus d'analyse des données HiKoB actualisant les tableaux de valeurs des centrales inertielles et le processus de calcul des angles. Lorsque je déclenchais une interruption matérielle via l'un des connecteurs GPIO du Raspberry Pi, lié à une routine d'interruption, celle-ci démarrait un cycle de modification de la stimulation. La stimulation appliquée étant la représentation de la valeur d'angle la plus récente disponible. Avec ce test de réactivité ne prenant pas en compte la latence angulaire introduite par les centrales inertielles, j'ai pu estimer la durée du cycle "récupération des valeurs d'angles disponibles et stimulation en conséquence" à environ 25-30 millisecondes.

Le test final de réactivité du système donne donc un résultat mitigé. En effet d'un coté les performances semblent prometteuses et conformes aux besoins, mais de l'autre, certains points bloquants nécessitent plus de temps pour investiguer, trouver et appliquer des solutions.

Conclusion

A l'heure de ce bilan, certains points restent encore à finaliser :

— la préparation des boîtiers contrôleurs et batteries pour disposer de deux kits de prototypage fonctionnel,

— le test du système complet sur batterie et donc les performances en situation réelle ainsi que l'autonomie effective du contrôleur.

Il semble également que l'ordonnancement du système soit très limité à cause du manque de possibilités de configurations lorsque le système est en cours de fonctionnement. Il aurait été en effet intéressant de pouvoir modifier les canaux de communications utilisés par les réseaux de capteurs et de stimulateurs lors de l'initialisation du système, ou de moduler les puissances d'émission des trames de communication sur les réseaux afin d'éviter des pertes liées à l'environnement et aux mouvements du patient.

Cependant, la majorité des objectifs du stage sont remplis :

— Le simulateur de contrôleur temps réel porté sur le système embarqué est découplé des parties intégration numérique et émulation des réseaux de stimulateurs. Cela permet donc à l'équipe de réaliser des simulations avec le matériel embarqué compris dans la boucle de test (Hardware In Loop).

— Les boîtiers qui ont été réalisés sont très compacts, optimisés pour le contrôleur et prêts à être utilisés.

— Il est possible de continuer le développement du contrôleur et d'ajouter aisément des fonctionnalités variées, grâce par exemple à l'utilisation du buzzer et des autres interventions extérieures disponibles.

— Le système peut être lancé à distance, avec son interface graphique, via des plateformes Linux ou Windows, sans nécessiter une connexion internet.

— La carte peut être utilisée comme base de prototype pour des projets ultérieurs nécessitant un système embarqué avec des capacités temps réel, grâce au noyau Linux modifié et patché.

Le contrôleur qui a été développé peut déclencher et superviser la stimulation, ainsi que la moduler en intensité et en largeur d'impulsion. Il permet aussi de traiter des interruptions extérieures pouvant provenir du patient, du chercheur ou du matériel utilisé conjointement avec le système. Sa réactivité interne semble satisfaisante mais il sera peut être nécessaire de le traduire en langage C++ afin de gagner en précision et en contrôle car le langage Python n'est pas réellement adapté au développement d'applications temps réel.

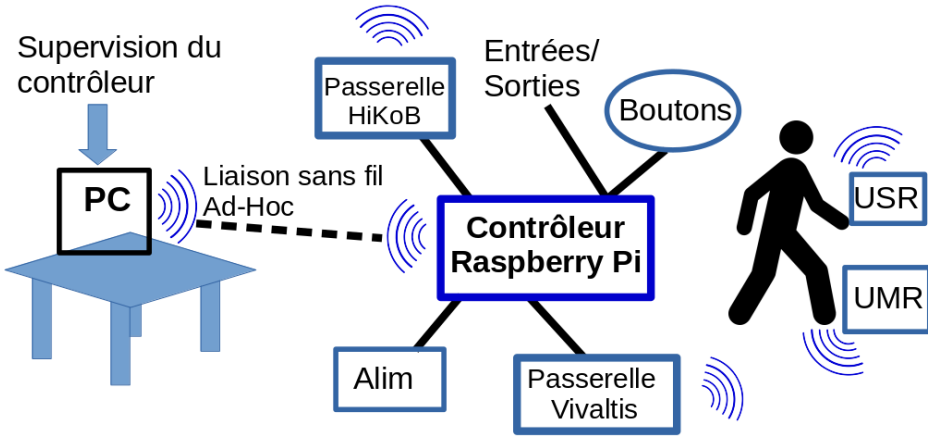


FIGURE 3.27 – Architecture du contrôleur développé.

Ce stage m'a permis de mettre en œuvre de nombreuses connaissances acquises pendant mes études à l'ENIB, à la fois dans le domaine de ma spécialisation "Electronique embarqué" mais aussi dans des domaines plus éloignés comme par exemple en Conception Assistée par Ordinateur.

J'ai aussi pu découvrir la gestion d'un projet dans sa globalité. J'ai été amené à utiliser plusieurs langages de programmation (C++/C, Python et Matlab), à poser le cahier des charges, à concevoir le système désiré, à sélectionner, à commander et enfin à préparer le matériel pour qu'il corresponde à nos besoins. Pour réaliser ces différentes tâches j'ai beaucoup appris, notamment en auto-formation dans les domaines suivants : FES, noyau Linux, Xenomai, patch RT_Preempt, Python et plusieurs de ses librairies, développement pour un fonctionnement temps réel, ordinateur carte unique (RPi).

Ce stage m'a également permis de vivre une expérience humaine et sociale en alternant le travail autonome et en équipe. Les échanges interdisciplinaires (Ingénieur qualité, doctorant en FES, chercheur spécialisé temps réel, chercheur chargé de recherche/chef de projet) ont étés particulièrement enrichissants. Ces collaborations m'ont amené à mieux me positionner dans ma fonction d'ingénieur.

Ce stage m'a enfin permis de confirmer mon goût pour la recherche médicale et particulièrement sur un sujet qui me passionne depuis longtemps, les neuro-prothèses. Mon projet personnel se précise et c'est en effet dans cette voie que j'aimerai poursuivre en tant qu'ingénieur ou doctorant.

Annexe : Tests de latences sur Raspberry Pi 3 B

Afin d'étayer les modifications de performances vous trouverez ci-dessous les résultats de tests de latences du noyau Linux 4.9.27 allégé et modifié avec le patch RT_PREEMPT rt16. Le test utilisé, Cyclictest, effectue des cycles endormissement-réveil et détecte le décalage entre la date de réveil demandée et la date effective du réveil du programme. Cette mesure est réalisée en boucle et dans plusieurs threads en parallèle, avec différents intervalles (colonne I) entre les réveils pour chaque thread. Toutes les mesures temporelles sont en μs.

Cyclictest est diponible sur le dépôt Git officiel à l'aide par exemple de cette commande :
git clone git ://git.kernel.org/pub/scm/linux/kernel/git/clrkwllms/rt-tests.git
Ou sur le site équivalent :
https ://git.kernel.org/pub/scm/linux/kernel/git/clrkwllms/rt-tests.git/

Le thread principal (numéroté "T : 0" dans la première colonne) est celui de référence. Les autres utilisent des intervalles progressivement plus grands. Les deux colonnes les plus à droite représentent respectivement "Avg" l'écart moyen et "Max" l'écart maximum enregistré.

Dans les deux images suivantes les tests ont pour référence un intervalle de 1 milliseconde (I : 1000 μs) et sont réalisés sur au moins 350 000 cycles, soit 350 secondes. Le mode performance est évalué sur la première image, l'horloge du processeur est donc de 1,2 GHz. Le premier test étant un témoin, il est effectué sans charge particulière sur les cœurs du processeur. Le suivant est réalisé en chargeant à 99% les cœurs, comme le montre l'onglet de monitoring adjacent, à l'aide des commandes :

1. cat /dev/zero > /dev/null
2. sudo ping -n -i 0.0001 localhost

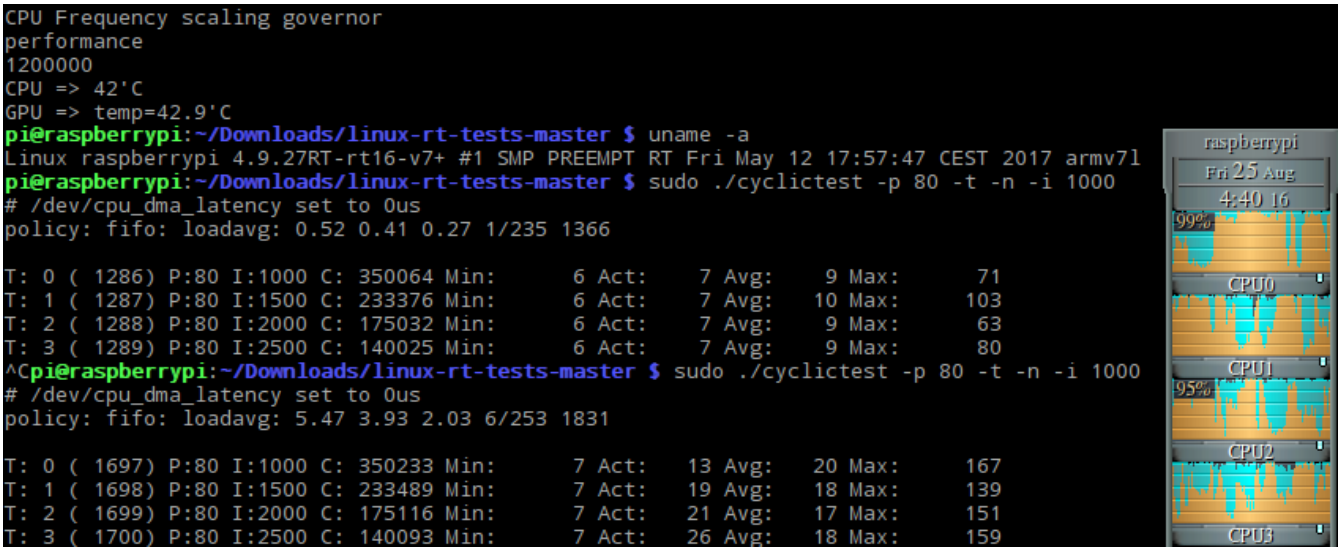


FIGURE 3.28 – Tests de latences sur noyau Linux 4.9 patché RT_Preempt. Témoin en haut, en charge dessous. Horloge CPU 1.2 GHz

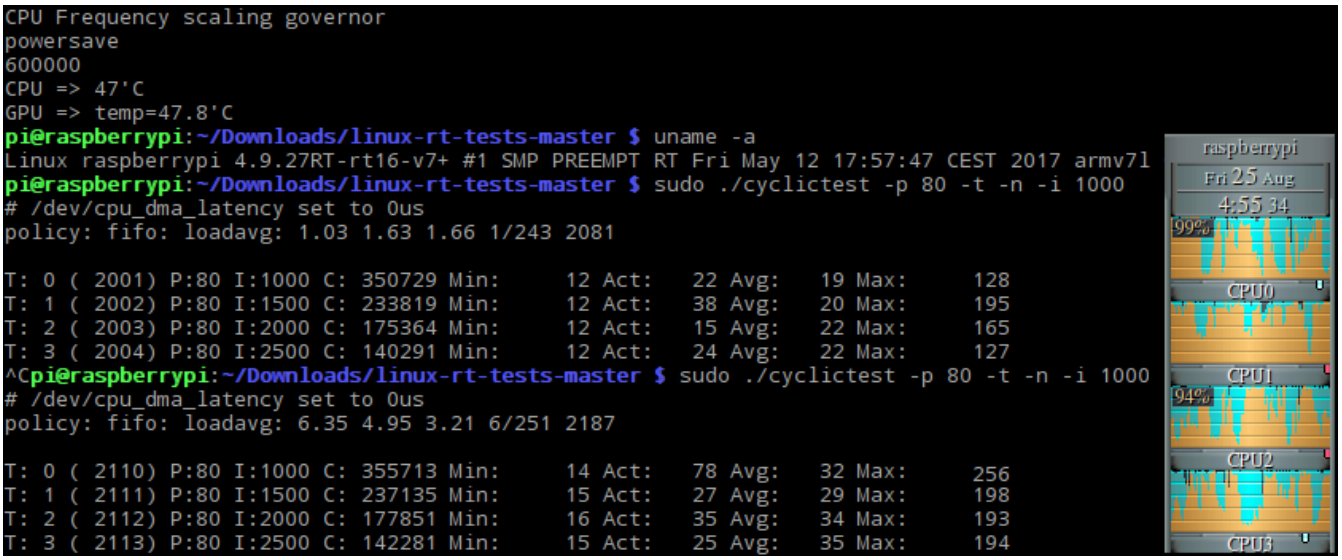


FIGURE 3.29 – Tests de latences sur noyau Linux 4.9 patché RT_Preempt. Témoin en haut, en charge dessous. Horloge CPU 600 MHz

Les tests suivants sont effectués avec le script de stress-test que j'ai créé, consommant un maximum de bande passante internet et entrainant un comportement plus instable du processeur. Le premier est effectué sur 500 000 cycles et le deuxième, plus critique, sur 1 500 000 cycles.

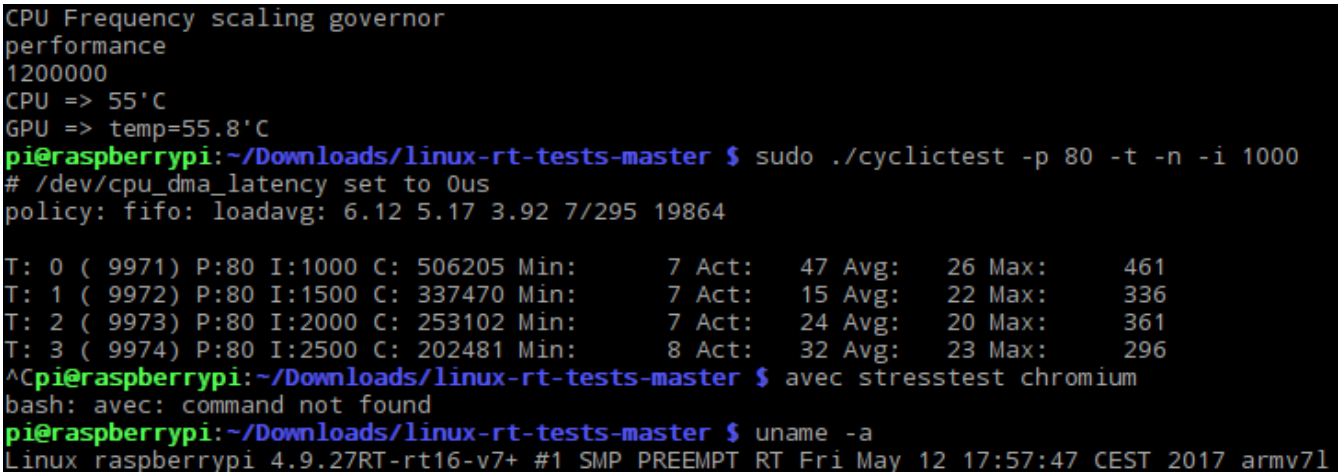


FIGURE 3.30 – Tests de latences sur noyau Linux 4.9 patché RT_Preempt. Stress-test de bande passante. Horloge CPU 1.2 GHz

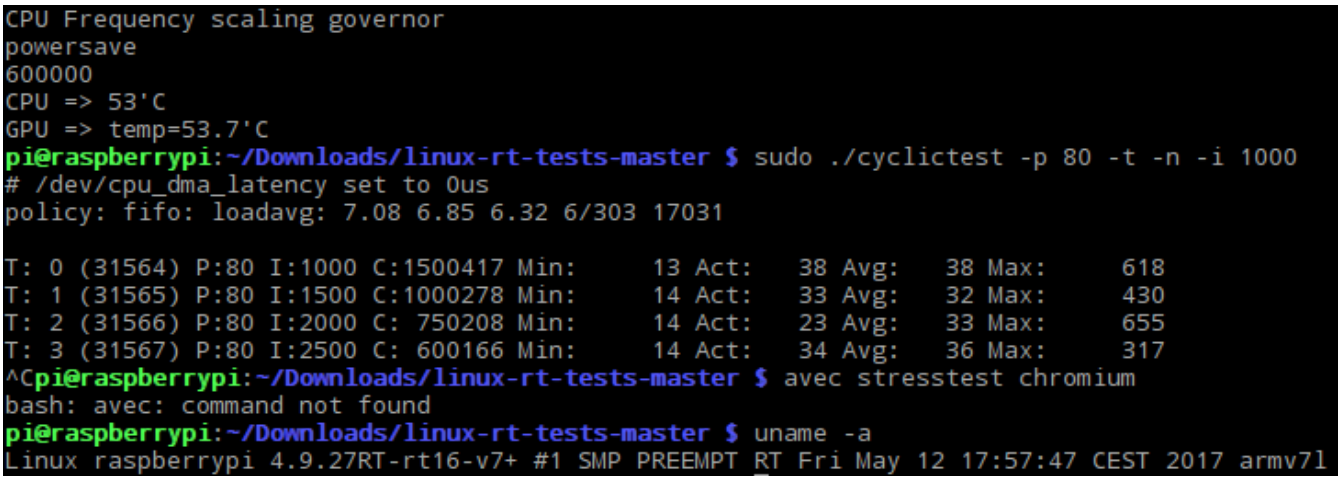


FIGURE 3.31 – Tests de latences sur noyau Linux 4.9 patché RT_Preempt. Stress-test de bande passante. Horloge CPU 600 MHz

Vous trouverez ensuite des diagrammes représentant la répartition des mesures de latences sur des tests équivalents aux précédents (Stress-test de bande passante). L'axe vertical, en puissances de 10, détaille le nombre d'occurrences, tandis que l'axe horizontal, linéaire, indique les valeurs des latences mesurées. Ces tests ont chacun été effectués sur une durée de 2 heures.

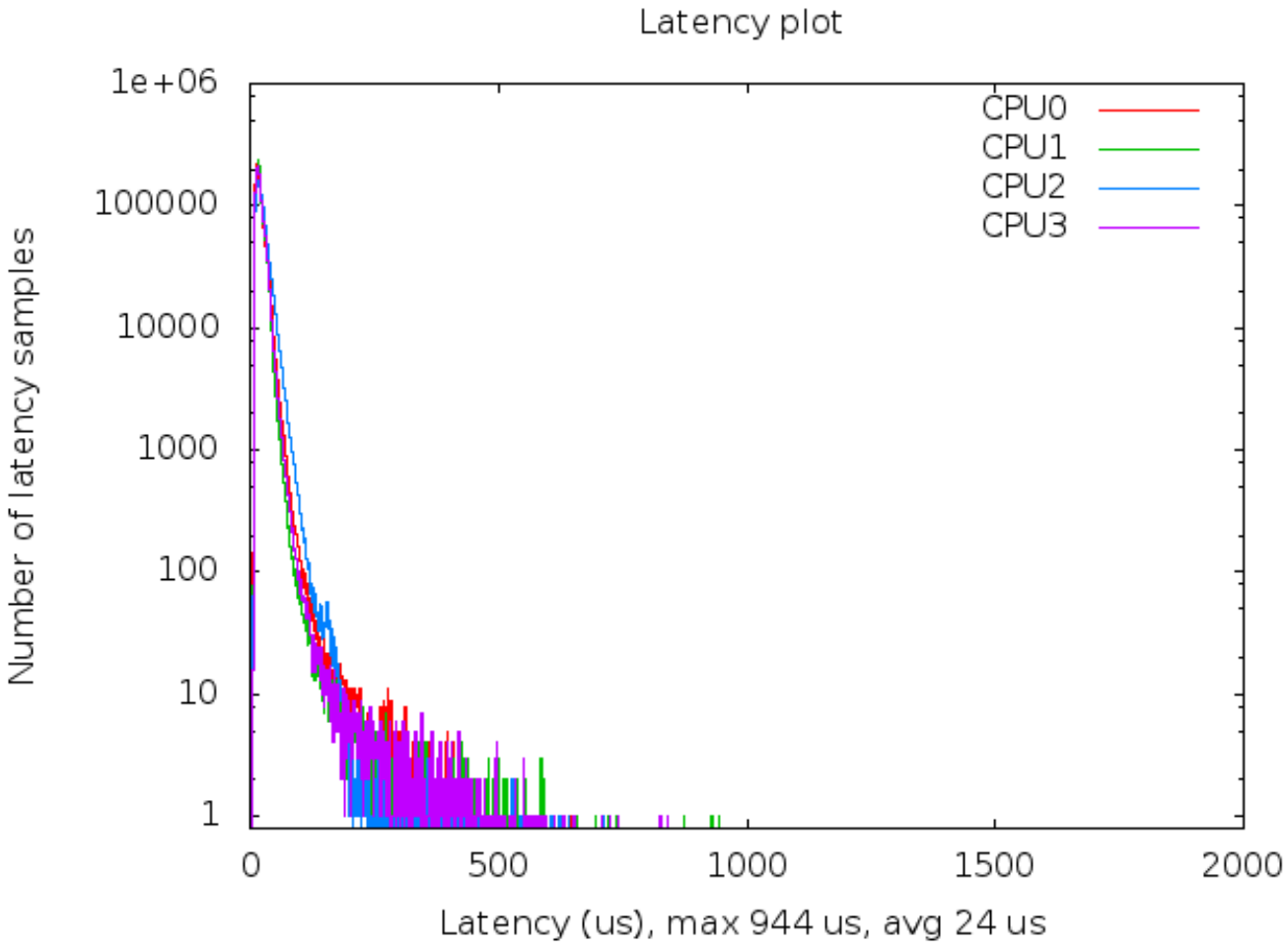


FIGURE 3.32 – Diagramme de latences sur noyau Linux 4.9 patché RT_Preempt. Horloge CPU 1.2 GHz

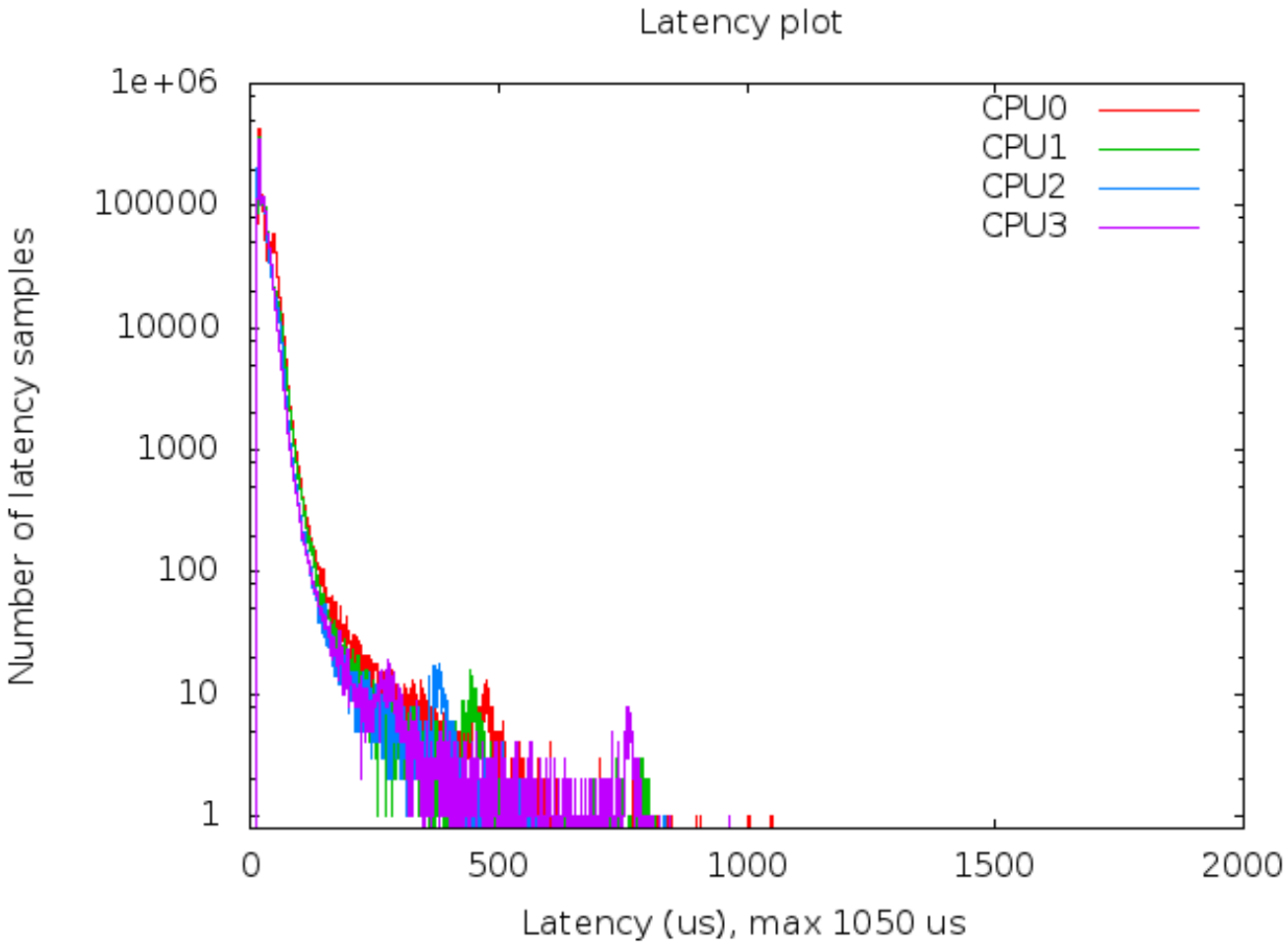


FIGURE 3.33 – Diagramme de latences sur noyau Linux 4.9 patché RT_Preempt. Horloge CPU 600 MHz